(4)

LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

MIT/LCS/TM-363

ASSISTING DESIGN GIVEN MULTIPLE PERFORMANCE CRITERIA

Dennis C. Fogg

August 1988

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) MIT/LCS/TM-363	5. MONITORING ORGANIZATION REPORT NUMBER(S) N00014-83-K-0125
6a. NAME OF PERFORMING ORGANIZATION MIT Laboratory for Computer Science 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research/Department of Navy
6c. ADDRESS (City, State, and ZIP Code) 545 Technology Square Cambridge, MA 02139	7b. ADDRESS (City, State, and ZIP Code) Information Systems Program Arlington, VA 22217
8a. NAME OF FUNDING / SPONSORING ORGANIZATION DARPA/DOD 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) 1400 Wilson Blvd. Arlington, VA 22217	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.
11. TITLE (Include Security Classification) Assisting Design Given Multiple Performance Criteria	
12. PERSONAL AUTHOR(S) Fogg, Dennis C.	
13a. TYPE OF REPORT Technical 13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Year, Month, Day) 1988 August 15. PAGE COUNT 63
16. SUPPLEMENTARY NOTATION	
17. COSATI CODES FIELD / GROUP / SUB-GROUP	18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) computer-aided design, artificial intelligence, linear programming, search, computer architecture, application-specific integrated circuit

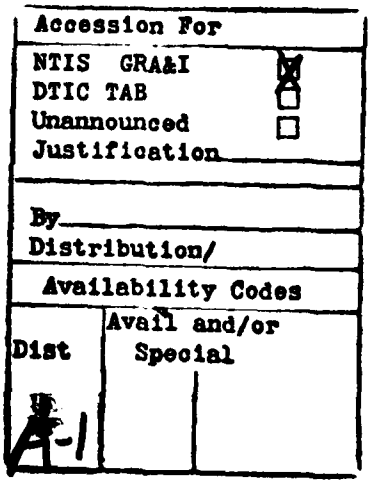
19. ABSTRACT (Continue on reverse if necessary and identify by block number)

A design system that accepts multiple performance criteria faces the problem of trading off one criterion for another. Understanding the user's preference about such tradeoffs is essential to producing the most desirable design. An argument is made against specifying preferences over all possibilities before the design process begins. The proposed solution reduces its dependence on the specification's accuracy by encouraging interaction between the user and the system. The system produces quick, high quality information aoubt realizable designs and performance tradeoffs between them. The user interactively evaluates, directs, and terminates the design exploration. Techniques called decoupled design, alteration strategies, and sample search are key elements in the implementation. Decoupled design generates a large number of designs efficiently (an alternative technique that uses linear programming is also presented). Alteration strategies are abstract descriptions of design modifications that assist in directly the design process and creating novel structures. Sample search is a framework to control

(continued on back)

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☑ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Judy Little, Publications Coordinator	22b. TELEPHONE (Include Area Code) (617) 253-5894 22c. OFFICE SYMBOL

19. search using domain specific heuristics. The design system operates in the domain of non-regular, application specific, signal processing architectures. The design process occurs in two phases: first, the architectural structure with uninstantiated operators is created then each operator's implementation is selected.



A

Assisting Design Given Multiple Performance Criteria

Dennis C. Fogg

July 26, 1988

A design system that accepts multiple performance criteria faces the problem of trading off one criterion for another. Understanding the user's preference about such tradeoffs is essential to producing the most desirable design. An argument is made against specifying preferences over all possibilities before the design process begins. The proposed solution reduces its dependence on the specification's accuracy by encouraging interaction between the user and the system. The system produces quick, high quality information about realizable designs and performance tradeoffs between them. The user interactively evaluates, directs, and terminates the design exploration. Techniques called *decoupled design, alteration strategies,* and *sample search* are key elements in the implementation. Decoupled design generates a large number of designs efficiently (an alternative technique that uses linear programming is also presented). Alteration strategies are abstract descriptions of design modifications that assist in directing the design process and creating novel structures. Sample search is a framework to control search using domain specific heuristics. The design system operates in the domain of non-regular, application specific, signal processing architectures. The design process occurs in two phases: first, the architectural structure with uninstantiated operators is created then each operator's implementation is selected.

Keywords: computer-aided design, artificial intelligence, linear programming, search, computer architecture, application-specific integrated circuit.

This research was supported, in part, by the Advanced Research Projects Agency of the Department of Defense under contract N00014-83-K-0125.

Contents

1 Introduction

Application specific, VLSI systems are significantly more efficient than their general purpose, microprocessor based counterparts. Unfortunately, they also demand much more design effort. This proposal investigates the use of Artificial Intelligence (AI) techniques to reduce the cost of exploring different architectures.

The key difference between this design system and most previous AI design systems is that the design specifications include multiple, interacting performance specifications in addition to the traditional functional specification. The ability to explicitly consider tradeoffs between various performance factors changes both the designed object and the process by which the object is designed. The system accepts a behavioral input describing the function of the architecture and a performance input describing the preferred tradeoff between performance and resources; its output is a set of possible architectures. The functional specification is described by a simple programming language which is easily translated into a dataflow graph. In the ideal case, the description can include constructs for looping, conditionals, and even non-recursive function calls. The system proposed here is more modest but should be amenable to scaling. Evaluating a system's performance typically involves many disparate factors including throughput rate, latency, chip area, power usage, technology, and pin count. The system will assist the designer in deciding tradeoffs among relevant factors. The user can influence the architectures by suggesting his own architecture or proposing a partial architecture and letting the system attempt to finish and optimize it. The system's domain is a niche of signal processing architectures in which computation is the bottleneck and calculation involves heterogeneous operators.

Related work in the literature is reviewed in the remainder of this section. The next section motivates the need for a new approach to multiple performance design and answers the need by presenting an overview of a solution. The following three sections describe in detail mechanisms that will be used to support the approach. Appendices contain additional detail.

1.1 Related Work

1.1.1 AI Techniques for Design

Design for Functional Goals The two primary models of the design process are: the refinement paradigm and the transformational paradigm ([Mos85] reviews both and discusses other issues in modeling the design process). According to the refinement paradigm, design is refining the specification into its component parts each of which is closer to a concrete implementation. Each part in turn is refined until every part of the specification has been mapped into a concrete implementation. The design history resembles a tree with the specification at the root and the implementation as the leaves. Note that component

parts can only affect parts that are descendants of it. Often the model is augmented with constraints, so communication can occur among parts in different branches of the tree. An example of the refinement paradigm is [MSS84]. [Roy83] expresses similar views on design except he describes the refinement process differently. He views design as a mapping from function to structure. The initial specification consists of a completely functional description of the design. The functional description is refined by substituting constructs that are both functional and structural. Remaining functional descriptions are refined until the implementation is completely structural.

The transformational paradigm views design as a series of transformations that convert the specification into an implementation. Unlike refinements, which expanded a single design component independent of other components, a transformation can replace any or all the design components. Thus, the transformational paradigm can express designs that the refinement paradigm cannot. Sometimes, the transformational paradigm also models the process of deciding which transform to apply. A hierarchy of goals is created, and when a goal can be satisfied by a transform it is applied. The design history has a tree of goals with transforms at the leaves. The implementation is created by transforming the specification by all the tree's leaves. Examples of the transformational paradigm are [Bal81] and [LM85].

Design under Performance Constraints The refinement and transformational paradigms concentrate on creating a design to satisfy a functional goal. This proposal emphasizes *performance design,* satisfying both a functional goal (what it should do) and performance goals (how well it should do it). There are important benefits to encoding performance preferences explicitly. A purely functional design system may create a design that uses too much area or operates too slow. In either case, the design is unusable. The user's only recourse is to tinker with the functional specifications and hope he can create a design that comes close to his performance specifications. Since the system created the design, it has the most information about it and is in the best position to modify it to meet the performance specifications. To date, only a few researchers have explored performance design.

Two projects have attacked performance design in the VLSI domain. [KP86] uses a STRIPS-like planner to determine the sequence of operations to apply. During the planning phase the design's evolution is represented by states and an associated set of logic statements describing the design. For example, the goal state might include the statement "(exist PLA-Layout)." The planner simulates operation application by adding and deleting logic statements. To execute a plan, operations are applied to the inputs to produce the actual design. Thus, the planner is not building the design, but rather controlling which tools are used. Planning and plan execution can be interleaved. The planner's search strategy is hill climbing: at the current node the most promising operation is applied where promise is measured by maximizing a weighted sum of performance measures. After the initial design

is produced, it is compared with the target performance. Subsequent designs are created by giving the operations different performance preferences. They are guided by interpolation to converge on the target performance. The user's performance goals consist of weights for 4 performance factors: area, speed, power, and design time. The other project is [BG86]. Its design model emphasizes that each refinement level has its own functional and performance specification. Each level also communicates to its parent and children. A level's performance specification can change depending on the resources handed down from its parent. Child levels can request more resources from their parent. Propagating resource allocations up and down refinement levels influences a design's performance, because different resource allocations create different implementations. Initially, the system produces designs at the extremes of the performance space. Interpolation creates new designs that are closer to the user's desired tradeoff ratio between time and area.

Kant's LIBRA program ([KB81, Kan79]) operates in conjunction with a transformational paradigm automatic programmer. LIBRA controls the application of transformation rules such that they create designs that are efficient and the design process itself is efficient. LIBRA uses a variety of techniques: it sorts decisions based on how influential they can be; it computes upper and lower bounds on partial implementations so search can be pruned by branch and bound; it explicitly allocates and obeys time limits put on the design of subgoals; and it uses a knowledge base to instantiate an implementation given the characteristics of the designed object and the amount of design resources left. LIBRA's approach for addressing multiple performance criteria design is to combine all criteria into a single evaluation function and maximize it. All three projects use a functional description to specify performance. The disadvantages of using a function as a user interface will be discussed in the next section.

Kowalski [Kow85] built a rule-based system to convert register tranfer level descriptions into architectures. Its architectural modifications include mapping several virtual registers and operators into one physical implementation and converting multiplexor structures into bus based interconnect. Since the program did not incorporate a design simulator, adjusting the design for performance was crude: users could specify limits on the number of operators allowed in designs (eg: constrain the system to produce a design with no more than 4 adders). Also, it was not built with a general model of design in mind, so the ideas cannot be used by design systems in other domains.

A new idea for design called *emergent criteria* is presented in [Nav87]. The assumption is that there are too many goal criteria for the user to consider initially, so the system will recall relevant ones when they are needed. His system creates a number of designs and ranks them along several goal criteria. Designs that cannot be dominated by other designs form a frontier of best designs for these goal criteria. The designs at the frontier as well as those just below the frontier are selected for further discrimination. The system has a

memory of previous case studies. Each case includes the winning design and the goals that were used to select it. If a goal criterion from an old case study can discriminate between the designs in the current situation, then the user is asked if he wants to use this criterion to discriminate the current designs. In its present state, the system may need the user to manually augment each design for the new criteria.

1.1.2 Silicon Compilers

Another related area of research is silicon compilers [Sou83, DR85]. Silicon compilers map hardware behavioral descriptions to layout. The input is a register transfer level description of an architecture. Silicon compiler proponents point out that this is a high level language because the input language's registers are virtual registers and do not necessarily correspond to real hardware. Several virtual registers may share one physical register, or if a virtual register does not need to store a value, it is deleted leaving only a wire. Unfortunately, this is the only architectural change the system can make automatically. If the user wants a different architecture to produce a different performance, he has to change the input description himself.

A key organizational difference between the system presented in this proposal and silicon compilers is that compilers have not integrated design evaluation into the design loop. In this proposal provisions are made from the start to allow the design component to be influenced by the design evaluator (simulator). The system can adjust the architecture based on the user's performance goals and the results of the evaluator. In addition, the proposed system has a much larger architectural vocabulary than a typical silicon compiler would.

1.1.3 Virtual Mapping Schemes

Another related area of research is the mathematical deduction of signal processing architectures. These methods will be referred to as virtual mapping schemes, because they map one mathematical description of an architecture into another. The work of Cappello and Steiglitz [CS84] is representative of these schemes. They start by mapping an equational description of a common signal processing computation into a recurrence relation. The relationships between the input and output variables are plotted in an n-dimensional space. There is one dimension for each index in the computation and an extra dimension representing time. The plot consists of points where computation occurs and lines for passing variables between computations. The plot's orientation relative to its axes implicitly defines an architecture. For example, all the computations in one time plane must occur before any computation in the next plane. Similarly, computations located along the same spatial index value will probably be implemented in the same processor. The diagram may not specify all the details of the architecture. New architectures are automatically deduced by rotating, translating, stretching, projecting, and swapping the axes of the plot, since

these transformations change the plot's orientation and shape. They result in new timing relationships, different numbers of processors, introduction of pipelining, and different orchestrations of data movements. The transformations can be represented by matrices.

Virtual mapping schemes have been successful in deriving known architectures and discovering new ones. They are a step toward formalizing architectural design. Nevertheless, they are severely limited in the types of architectures they can manipulate. They have concentrated on signal processing architectures, because those computations have very regular structure. For example, Cappello and Steigltz illustrate architectures for matrix times vector, convolution, matrix product, and matrix transpose. Extending the scheme to nonregular or even slightly irregular computations is not clear.

2 A New Approach

2.1 The Difficulties in Specifying Multiple Performance Preferences

2.1.1 Representing Preferences by Constants

At first glance the method of specifying performance seems analogous to specifying functional behavior. That is, the user simply dictates the functional operation and the performance characteristics of the design to the system. The problem with this approach is the performance specifications are completely divorced from what is physically realizable. There is no reason to believe that a chip can be built to the user's performance specifications other than the user wants it that way. In fact, if such a system did exist, the clever user would demand that designs take zero time and occupy zero area.

Typically, one of two situations occurs: It may be impossible to build a design that meets the specifications (over constrained) or there may be many designs that not only meet the specification but do better (under constrained). If the design is over constrained, a typical system would fail and give no hint about how to relax the specifications to create a workable design. If the design is under specified, the system does not know which of the better solutions to choose it because it does not know the user's preference on trading off performance dimensions. More importantly, a system that has found an under constrained solution may not search for better ones, so the user would not be told that better alternatives exist.

2.1.2 Multiattribute Preference Functions

An alternative technique is to bundle all the performance factors into one function and optimize it. This solution is often used when coarse comparisons are required. But when a function is used to choose the best design, finding its form and coefficients is critical and difficult. Even with two performance factors, it is difficult to specify a mathematical

relationship that precisely reflects the user's tradeoff preference. He may have a feel for the general nature of the tradeoff or he may know which tradeoffs are obviously undesirable. But, it is neither obvious nor natural for a designer to know the number of squared microns he is willing to trade for a nanosecond speedup. This is aggravated by the lack of context to make the decision. The function must be an exact description over all possible values of the 2 parameters. Even if the user actually produces a preference function, it's not clear that he understands the implications of the new statement. Since this is not how users naturally represent their tradeoff knowledge, statements made in this language are unreliable. The preference function guides the creation of designs, so if it is inaccurate it guides the system in the wrong direction. There has been some work on specifying perference functions using properties of the desired function ([Wel85]), but they provide little assistance in guiding the design process.

The performance preference problem for a VLSI system is further complicated by the number of factors that might be included in the function, including: latency, throughput, area, pin count, design cell count and availability, power, routing and layout issues, control conplexity, I/O protocal, etc. This list is not exhaustive; each application may have factors that are unique to that design. One might argue that the number of factors could be reduced by considering only the "important" ones. The problem is that any factor can be "important" if its influence grows large enough. Since the user does not have the design in front of him, he does not know the magnitude of the factors and thus does not know which ones are important. He has to specify when each factor becomes important and how to trade it off when it is important.

When the user interface is a function, the user has to give a complete characterization of his preference for all possible cases even though most situations will not appear. This is an enormously difficult task that is certainly extremely error prone. The smart user will be wary of a system that produces a "best" design according to his performance specifications. Design assistants don't know what the best design is so they should not concentrate on producing a single best design. The difficulties of specifying multiple performance preferences will be referred to as the problem of unreliable performance specifications.

2.2 Framework for a Solution

There is fundamental barrier between the design system which cannot know what the user wants and the user who does not know what designs are possible. A potential solution is to create cooperative interaction between the two parties such that each gradually finds out more about what the other knows. Given this view, the system's responsibilities change substantially. Instead of asking the system to do the impossible (produce an optimal solution given multiple criteria), the system should help the user explore alternative designs and provide information about tradeoffs among performance dimensions. The user and the

design system are in a feedback loop. The system creates and measures designs, and the user evaluates the solutions and guides the search process. The information passed between the user and the system includes not only the designs but also possible tradeoffs between performance dimensions, the size of the search space, the method of producing the designs, and ways to optimize the designs.

Incorporating feedback sharply contrasts systems that rely on accurate functional specifications. It is convenient to view the design process as consisting of 3 entities: the user, the specification, and the implementation. A system that assumes it receives accurate functional specifications from the user simplifies the interface between one pair of entities at the expense of another. Its specification/implementation transformations can take advantage of a well-defined evaluation criterion, a limited search space, possibly a termination condition, and a wealth of knowledge on search and optimization. Unfortunately, the user/specification interface shoulders the unreasonable burden of precisely quantifying the user's preferences. By introducing feedback from the implementation to the user, the design process is less dependent on the accuracy of the user/specification interface and more likely to identify a preferred design. The design system, however, has less direction from the specifications. New techniques are needed to prune the search space and to make each search step more efficient.

There has always been interaction between user and design system, but previous systems have not been designed to explicitly take advantage of it. The approach advocated here anticipates its role in the partnership and capitalizes on it in several ways. First, tradeoffs between real designs can be efficiently produced and are valuable to the user. Users prefer choosing among actual designs rather than specifying tradeoff preferences over all possibilities. Second, sometimes it is more important to get quick information even at the expense of compromised solution quality. Given limited resources, a system that always produces the best designs is not necessarily producing the best information. If the user is only interested in whether he is "in the right ballpark" then a system that produces less optimal solutions faster is preferred. This situation is often true in the initial phases of design when the user is exploring the space. In later phases, the user will probably want high quality designs and is willing to wait for them. The design system should be flexible enough to operate between these extremes.

Third, the user/system interaction exposes the dual role of performance specifications. Previous design systems have a single description of the desired performance. Those systems direct the search for designs according to this goal and return the solution that fulfills it best. Interactive specification forces a distinction between using a performance specification to guide the search and using it to select the most desirable design. User interactions generate many different performance goals all of which are only an approximation of the user's actual preference. The interactions direct the system to explore different directions. Selecting the

most desirable design is done by choosing from actual designs instead of maximizing an abstract description of preference. Thus, performance goals in this project are similar to those in previous systems in that the system strives to achieve them, but they are different in that the system does not assume it has succeeded when it has achieve them.

2.3 Project Goals

This proposal is more than an outline of how to build a VLSI design tool or how to integrate AI techniques into a design tool. The underlying goal is to investigate how computers can assist (and possibly alter) the design process of real world objects. The main results should be ideas on the nature of a human/computer interaction for design exploration and techniques to implement them.

The general problem of assisting design is too broad for one thesis. This project explores the design of objects whose specifications contain interacting and conflicting performance preferences. The thesis proposed is:

> The design of objects with multiple, interacting performance specifications can be facilitated by encouraging feedback between the user and the computer. The design system produces quick, high quality information about realizable designs and performance tradeoffs between them. The three techniques of decoupled design, sample search, and alteration strategies are useful in implementing such a design system.

Other design automation projects have emphasized the role of the computer as an assistant, one that helps the user do the mundane chores and leaves the hard things for the user ([RSW79]). The system advocated by this thesis takes a more responsible role. As a colleague the system is providing a service that complements the user's skills. It extends the designer's capabilities by searching parts of the design space more quickly and thoroughly.

The design methods of the system should be explicitly encoded and program accessible; there are several reasons for this. First, the explicit identification of design techniques will add to the knowledge of the design process enabling future design programs to use a library of design techniques instead of reinventing the same. Second, general design techniques lead to the development and use of tools that speed up the development of other design systems. Third, if the design techniques are explicitly encoded and accessible by the program, then the system is not limited to making decisions about the designed object. It can expand its domain to that of making decisions about which design technique to use (trying alternative design strategies if the current one is faltering). Finally, since the techniques are known to the program, it can use them in its explanation of the designed object. It is often necessary to refer to the design process to explain the designed object. For example, if the user wants

to be convinced that the system's design is of high quality, one answer would be to discuss the design technique in general and this instance in particular.

2.4 The Domain

The proposed design system is not intended to be used for all signal processing algorithms. Many popular algorithms consist of the regular application of simple operations (eg: FFT, FIR filters). The best tool for these algorithms is either special purpose design programs or the virtual mapping methods described in Subsection 1.1.3. The tool proposed in here works for a small niche of application specific, signal processing where the computation does not have regular structure or symmetry that can be easily exploited through a regular architecture. The algorithms should be complex enough for a large variation in architectures but simple enough that a computer program can derive new architectures. Often the computation is complicated by unusual iteration patterns and interactions between heterogeneous operators. Four such algorithms are described in Appendix A:

- QR: an algorithm to find the eigenvalues of a restricted type of matrix,
- HH: an algorithm to solve systems of linear equations using orthogonal methods,
- EV: calculates the r largest eigenvalues, and
- CG: an indirect method to iteratively find the solution to a system of linear equations.

The design system assists in the high-level exploration of architectures. Its implementations are described at the register transfer level. Fabrication technologies, clocking schemes, and other implementation issues below the register transfer level are summarized by the performance of the primitive functional blocks and their interconnection.

2.4.1 Terminology

It might be helpful to introduce some terms for the different types of architectures that will be referred to. The input to the system is typically parsed into a *dataflow graph* which is an implementation independent description of the input computation. A *direct map architecture* is the simplest implementation of a dataflow graph: hardware operators are created for each graph node and wires are created for each graph arc. The direct map architecture is the fastest and most parallel implementation of the graph. The architecture can be pipelined and/or some operators can be shared into one physical operator, but this will still be considered a direct map architecture. One distinctive feature of this architecture is that there are no global buses; connections between operators are dedicated communication lines. This is in contrast to the *3 bus* and *1 bus architectures*. The 3 bus architecture has a memory and a processor. Two buses bring data from the memory to the processor

and one bus returns the results; the 1 bus architecture has a single bus doing all the data transfer. The memory and the processor parts of the architecture are not necessarily simple units. There may be several heterogeneous processors and/or memories. The *standard 3 bus architecture* has a single uniform memory feeding a multi-functional ALU.

Another dimension to consider is the number of processors in the architecture. A *multiple processors architecture* has more than one processor and those processors have the potential to run in parallel. The "processors" in a multiple processors architecture are independent hardware units that can do computation and can store results [1]. A typical multiple processor architecture would have n 3 bus architectures running in parallel working on different parts of the dataflow graph. Although the multiple processor architecture would work well on algorithms with regular structure, these problems are not considered in this project because they are best handled by virtual mapping schemes.

2.5 Overview of the Design Methods

The philosophy of this project is that the barrier between the user and the design system can be assailed by encouraging close interaction and feedback between the user and the system. The system contributes by producing high quality designs quickly and providing information on performance tradeoffs. The user combines this information with his goals to guide the search. The interaction is initiated when the user specifies his performance goals and the system returns designs that try to meet them. The user can modify his request after seeing a few designs and the system will produce designs targeted for his modified goal. Alternatively, the user can find local improvements in a set of designs by sending them to the local optimizer. The interaction continues until the user is satisfied.

The design process is split into 2 phases. The first phase, called architectural modification, transforms the user's specification into an architecture with abstract operators. Abstract operators have their function specified but not their implementation or performance. The second phase, operator selection, instantiates the operators so the performance of the overall architecture can be ascertained. A technique called *decoupled design* takes advantage of the fact that each abstract operator can be instantiated independent of the others. Using many combinations of operator instances, decoupled design produces lots of designs each of which operates at a different point in the performance space. The designs cover a wide range of operating performances, so the user can select a design that is close to his desired performance. Moreover, the group of designs gives the user an idea about tradeoffs between performance dimensions.

If the designs do not reach the user's performance goals or if the user wants to explore alternate designs, transformations can modify the architecture. Some sets of transforma-

[1] A pipelined, direct map architecture could be considered a multiple processor architecture, but it is a special case and is more descriptively referred to as direct map.

tions have a common intent; *alteration strategies* are abstract descriptions that capture that commonality. Alteration strategies allow the system to reason about the transforms as a group, justify a transform to the user, and assist in conflict resolution. The alteration strategies in this system are: refining structures into more detailed or concrete ones, physically sharing two structures, interleaving the details of two structures so they operate faster, and recognizing when a more efficient structure can be used.

The process of transforming the functional specification to a fully instantiated architecture includes many opportunities for choices. The series of choices can be thought of as a search tree. To avoid searching the entire space, traditional search methods must evaluate non-leaf nodes. Only leaf nodes in this space can be evaluated accurately, so the design system will use a technique called *sample search*. Instead of searching the entire subtree under a node, a few alternatives are sampled. Heuristics guide the sampling process. Assuming the samples can ascertain the design quality, the node can make more informed decisions about which alternatives to search more thoroughly.

The design system produces thousands of designs and plots their performance characteristics. The user can examine the schematic of his favorite design. He can explore other designs by revising the performance goals or trying another functional specification.

3 Operator Selection

This phase of the design assumes the abstract operators are fixed in an architectural structure and the only remaining decision is choosing operator implementations. Two solutions will be discussed. The first, called *decoupled design*, creates a very large number of designs but uses relatively few computational resources. Each design differs in detail from the others so the collection supplies the user with designs that cover a large swath of the performance space. The mass of designs is useful for finding a design at a particular point in the performance space as well as for displaying feasible tradeoffs between performance factors. The space of all possible decisions is huge, so heuristic methods shrink the space. The second solution is to use the simplex linear programming algorithm. By encoding operator selection with a linear objective function and linear constraints, optimal operator assignments can be found.

3.1 Disconnected Decisions

The general methodology is to find some computational leverage point to assist performance design. The aspect this proposal intends to exploit is *disconnected decisions*, a design decisions that can be made independently from other decisions and can be easily incorporated into the design regardless of the decision. It is independent in that it does not affect the functionality of the design; it may influence the performance. The set of possible design

choices are the *instances* of the disconnected decision. In this domain the disconnected decisions are choosing the operator implementations. The direct map architecture provides a good illustration. The decision about the operator implementations does not affect any other design decision and the method of incorporating the decision into the rest of the design is the same regardless of the choice. For example, if the system decides to use add-shift multipliers for every multiplier, the system could still choose any type of adder or could pipeline the architecture in any way. Moreover, the process of assessing the performance of the architecture would be the same for each implementation. Analogously, the decision about the processor implementation in a 3 bus architecture is a disconnected decision for the same reasons.

3.2 Decoupled Design

The main idea in decoupled design is to take advantage of the independence of disconnected decisions so a large number of designs can be produced efficiently. When a design contains 2 independent disconnected decisions the set of designs produced is the cartesian product of the disconnected decision sets. These designs are all different and occupy different positions in the performance tradeoff space. For example, in the direct map architecture the adder implementation is independent of the multiplier implementation. Instead of creating a design by a selecting particular adder and multiplier, the system creates a whole family of designs by using each adder/multiplier pair.

It is not enough for design creation to be efficient; design evaluation must be efficient too. Design evaluation does not necessarily mean the system has to resort to time consuming simulation. An alternative method is to produce parameterized functions to describe the architecture's performance and then instantiate the function with instance values. Looping through functions is much cheaper than simulating all the designs. The system can only derive an architecture's performance behavior using this method. The functional behavior is guaranteed because the system starts with a functionally correct architecture and changes it with truth-preserving transformations. This is a common technique in AI design systems.

Sometimes the normal design process prevents decoupled design from being used because design decisions are dependent on one another. Often decoupled design can be accommodated by

1. altering the design process whereby it creates good but not optimal designs,
2. apply decoupled design, and then
3. select designs near the user's desired performance point and repair the sub-optimal decisions introduced by the altered design process. This phase is performed by the local optimizer, which is described in section 3.4.

The pipelining process is an example. The optimal algorithm for pipelining a direct map architecture is: given the pipeline cycle time, walk up from the graph output inserting registers when one pipeline cycle time has been covered. This algorithm will produce an architecture with the prescribed cycle time and minimum latency. The problem is that it depends on the operator execution times making pipelining depend on operator selection. So a decoupled designer cannot use the optimal algorithm and is forced to settle for less. There are a number of ways to pipeline under the decoupled design conditions. One way is to use nominal values for operators and place the pipeline registers using the optimal algorithm. Another way is to look for regularities in the graph. If it notices a pattern then put pipeline registers around it and then pipeline the rest of the graph using the throughput time of the pattern as the standard. A third approach is not trying to pick the best partition the first time. Instead, the system can try several placement combinations and pick the best. A technique to support this is described in section 5. None of these methods will yield the best pipeline register placement but often they will be good enough to get an approximation of how that architecture will perform. A later process can jiggle the registers into a more optimal placement if the user decides he is interested in it.

Decoupled design attacks performance design by generating large numbers of designs quickly. Other approaches to multiple performance criteria modify a design to move it along various performance dimensions searching for the optimal tradeoff. They fall prey to the problems of unreliable performance specifications. Decoupled design does not force the user to give a complete characterization of his performance preferences for all possible cases. The system produces a large set of designs and the user simply picks the design that is closest to his ideal. In addition, the mass of designs provides information about tradeoffs between performance dimensions.

3.2.1 Reducing the Enumeration Space

The major stumbling block of decoupled design is that the enumeration space can be huge. The most obvious way to combine instances from different disconnected decisions is to try all combinations. This type of simple enumeration works fine for small numbers of decisions and instances; however, as the number of possibilities grows, the number of combinations grows exponentially. The key difference between decoupled design and simple enumeration is that decoupled design explicitly represents and considers the size of the space before it starts. Disconnected decisions are sets and decoupled design is combining sets into cartesian products, so it is easy to compute the size of the potential space. The decisions about how to reduce space are more complicated. Several heuristics have been identified to reduce the enumeration space. The difficult part is how to select and instantiate heuristics such that they shrink the space sufficiently and push the designs toward the user's performance goals. The current plan is to use domain dependent, knowledge based methods to make

these decisions. Heuristics currently being considered are listed below. The first set reduce the number of instances in each disconnected decision:

1. Remove instances that are dominated in all performance characteristics (this has already been done for multipliers and adders in this system)
2. If 2 instances have very similar performance use only one, especially in the first pass of the design process
3. Use fewer instances in each disconnected decision. There are various ways to prune instances. For example, when the design process begins, thin out instances but keep the dynamic range. In the later stages aim towards a particular performance range (eg: fast implementations).
4. Use knowledge based inference and information about the variation of other disconnected decisions to narrow the number of instances.

The second set reduce the number of disconnected decisions:

1. Do empirical testing to find out which disconnected decisions are less influential and bind those decisions to single values.
2. Do symbolic analysis of evaluation function to deduce which disconnected decisions are less influential and bind them to single values. Look at the range of instances and the relative weighting of disconnected decisions when they are combined.
3. Take 2 disconnected decisions of the same "type" that are independent and bind them together to form 1 disconnected decision. For example, if the implementations of 2 adders are independent, bind them so that they always have the same implementation. One heuristic is to bind all disconnected decisions of the same "type" together (eg: bind all adders to be the same). This is often one reasonable way to re-configure decisions but it is not the only way. Consider an architecture that has n operators where $n-1$ operators are used in a similar way but the n^{th} is used in a different way. Then, the n^{th} operator should not be in the same disconnected decision as the others.
4. Bind 2 decisions of different types by pre-enumerating the sensible combinations. For example, match fast multipliers with fast adders and likewise for slow operators. Maybe expand the matching by associating a fast multiplier with several fast adders. This reduces the $n \times m$ combinations to max(n,m) combinations.

3.3 An Example

A simplified, experimental version of decoupled design has been implemented. The intent is to test decoupled design on simple cases, gain experience, and find limitations. The current version works on very restricted architectures: no loops, no conditionals, and neither the operators nor the architecture is pipelined. Multipliers and adders are the only functional units used. Table 1 lists the performance characteristics for 16-bit implementations of each. Although all architectures in this example are 16 bits wide, it would be straight forward to

Operator	Time (ns)	Area ($10^3\mu^2$)
ripple carry adder	99	137
carry lookahead adder (CLA)	40	278
conditional sum adder	31	357
add-shift multiplier	644	470
shift by 3 non-overlapped bits multiplier	521	743
shift by 4 non-overlapped bits multiplier	434	880
shift by 6 non-overlapped bits multiplier	397	1154
shift by 8 non-overlapped bits multiplier	335	1428
Baugh-Wooley array multiplier	198	2083
Braun array multiplier	124	2371

Table 1: Library of 16-bit adders and multipliers.

parameterize the operators to create architectures of different widths.

Figure 1 shows the dataflow graph used in this example. It has the same structure as the QR algorithm in Appendix A (Figure 15) except the operators are multipliers, adders, and no-ops. If each operator was free to take on any implementation there would be over 352,000 possibilities. The system assists by helping answer the question: what is the best operator assignment for the user's preference of time verses area? Decoupled design's solution is to produce lots of choices for the user. One issue is how to control the enumeration to catch most of the good ones.

The approach is to create an enumeration language to specify different ways operator combinations can be generated. The current language is a list of terms. Each term in the outer most list defines an operator binding. For example, the expression:

```
((m8 m9) (np1 (np7 -1 1)) (m3 m4 (m2 -6 0)) (a6 1 2))
```

has 4 outer terms. Terms like `(m8 m9)` bind the operators `m8` and `m9` to be the same implementation. If `m8` is an add-shift multiplier then so is `m9`. The `(np1 (np7 -1 1))` term uses an implicit time ordering among the operators. `np7` is bound to be one slower, the

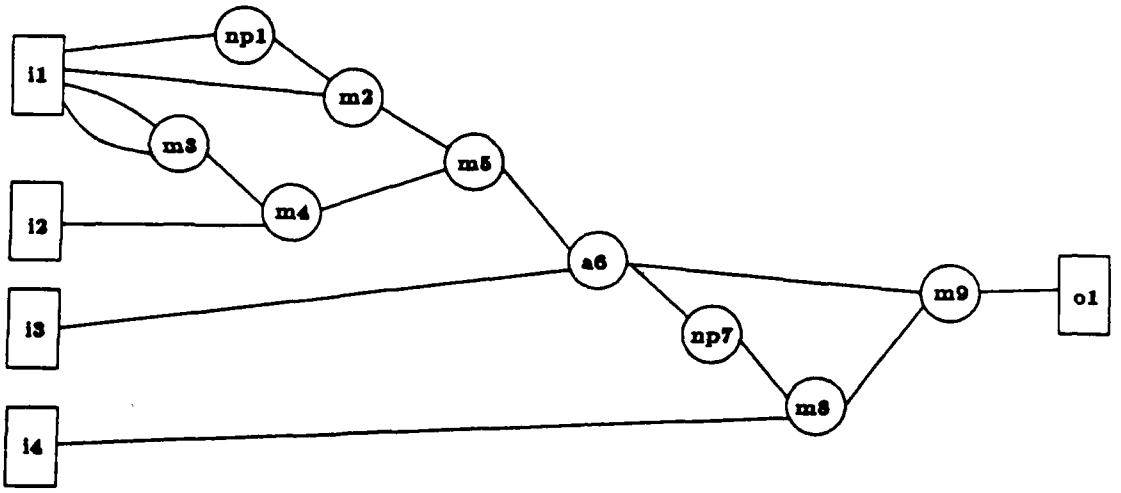


Figure 1:

same, and one faster than np1 for each instance of np1. These two terms can be combined as in (m3 m4 (m2 -7 0)) where m3 and m4 are bound together and m2 is bound to be all the implementations slower than them (because there are 7 multiplier implementations). Absolute assignments can also be made as in (a6 1 2) which binds a6 to the fastest and second fastest adders. Operators not mentioned are free to be any implementation.

The current language evolved from simpler constructs. The methodology has been to create a language, use it to generate designs, and examine the results. Two types of observations are sought: finding which combinations produce good designs and inferring combinations that may produce good results but which cannot be expressed. The latter motivates a change in the language.

In this prototype stage, the user controls the enumeration generation via the language. A reasonable expression for Figure 1's architecture is

```
((m5 m9) (m3 m4 (m8 -2 2) (m2 -6 0)) (a6 1)).
```

The reasoning behind the expression is heuristic. If there are 2 operators of the same type in series, then when they are bound together they act like an operator that has twice the delay. This is an easy way to reduce the space without making too much sacrifice. This accounts for (m5 m9) and (m3 m4). m2 operates in parallel with 2 multipliers, so its implementation will certainly be slower than theirs. Bottleneck operators should have fast implementations

which accounts for a6. Ideally, m8 should vary independently but that would make the enumeration space too large. It is set to vary such that the operators on the bottleneck path are kept in the same range. The no-ops have only one implementation to choose from. As more experience is accumulated, a subsystem will be developed that controls the enumeration automatically. This system will likely be heuristic and may be implemented as a rule based system.

Given the Figure 1's architecture and the associated enumeration expression, the decoupled design system will produce performance plots of the designs. Parameterized time and area functions are constructed from the architecture. The enumeration expression is translated into a set of nested loops with the time and area functions at the core. The decoupled design system compiles and evaluates the loop expression to produce the plot in Figure 2. The points can be pruned by keeping only the ones that are not dominated by other points (i.e.: keep points that are better in time or space or both). Figure 3 has pruned away designs that are dominated.

The plot can be used as part of a design style that differs from the traditional approach. A typical strategy in AI programs is to plan, infer, or reduce the difference towards a single goal. As soon as one solution is found, the problem is solved. The more realistic design process has multiple goals to meet. Decoupled design attacks the multiple goals simultaneously by a producing large set of different architectures. The volume of designs supports a more flexible design style. For example, the fringe of best designs gives the user a sense of the global tradeoff between time and area which might be used to do "what if" studies (what if the user had this much extra space, how much time would it buy?). The global tradeoff might actually surprise the designer and suggest opportunities. There might be a highly efficient design very close to the user's original performance preference and he may be willing to adjust his performance to take advantage of it. But this can only occur if the system finds the opportunity and tells the user.

Constrast the interaction encouraged by decoupled design with systems that specify performance using constants or functions. Constant performance goals are either never met (over specified) or are met with sub-optimal designs (under specified). Functional performance goals force the user to make exacting specifications about design that have not been created yet. Decoupled design allows the user to choose among concrete designs and to redirect the design goals as designs are being created.

Decoupled design has been able to control the exponential growth of enumeration. The 352,000 possibilities have been cut to 812 and were produced in 25 seconds on a Symbolics 3650. A significantly larger number of design can be outputted without incurring much time penalty: a less constraining expression that took 50 seconds produced 7200 designs. Experience with several other architecture of approximately the same size has confirmed the tractability of decoupled design. These result encourage further study in two areas: working

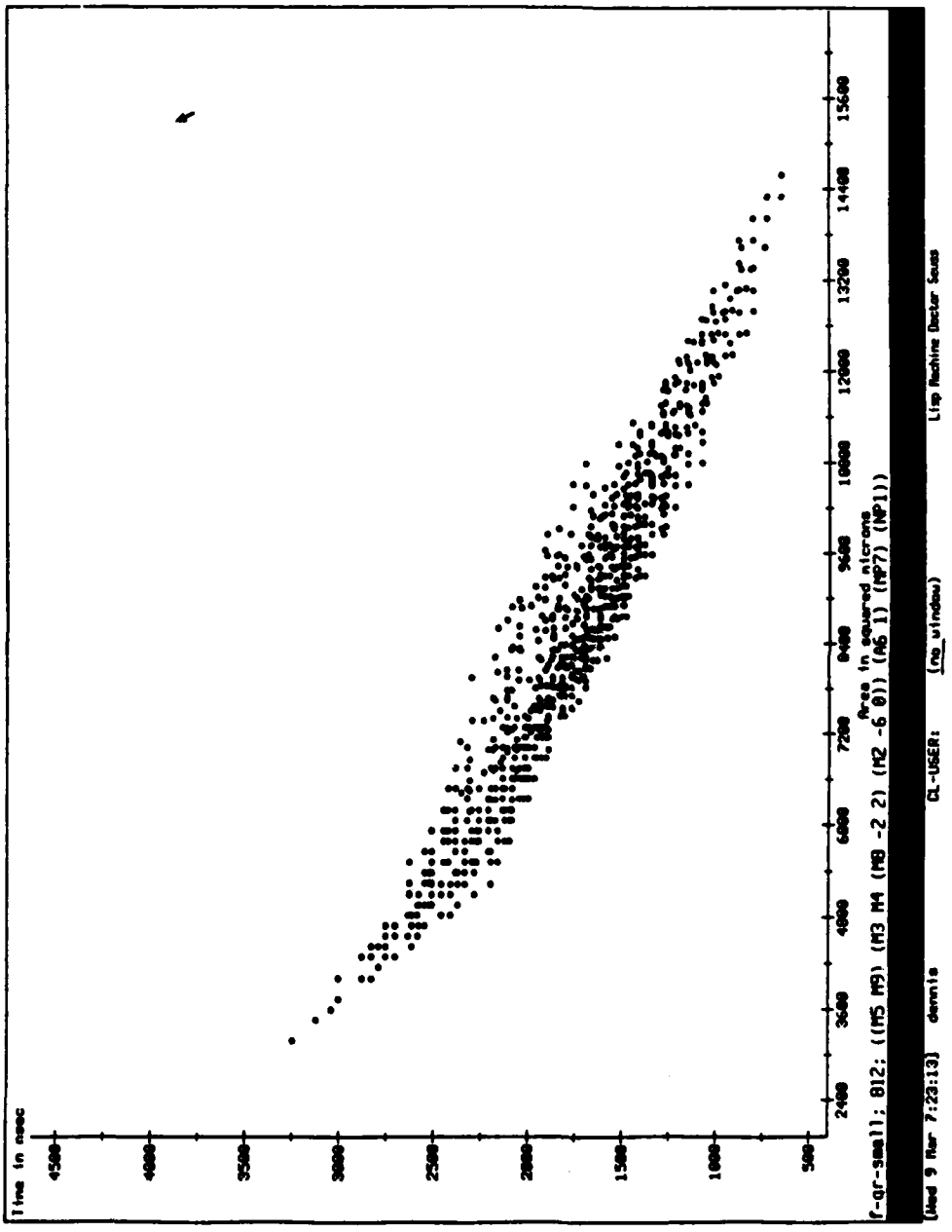

Time in msec
4500
4000
3500
3000
2500
2000
1500
1000
500
2400
3200
4000
4800
5600
6400
7200
8000
8800
9600
10400
11200
12000
12800
13600
14400
15200
15600
Area in squared microns
[-qr-small: 812: (((M5 M9) (M3 M4 (M8 -2 2) (M2 -6 0)) (M6 1)) (M7) (M1))
[Wed 9 Mar 7:23:13] dennis
CL-USER:
(no window)
Lisp Machine Doctor Seuss

Figure 2:

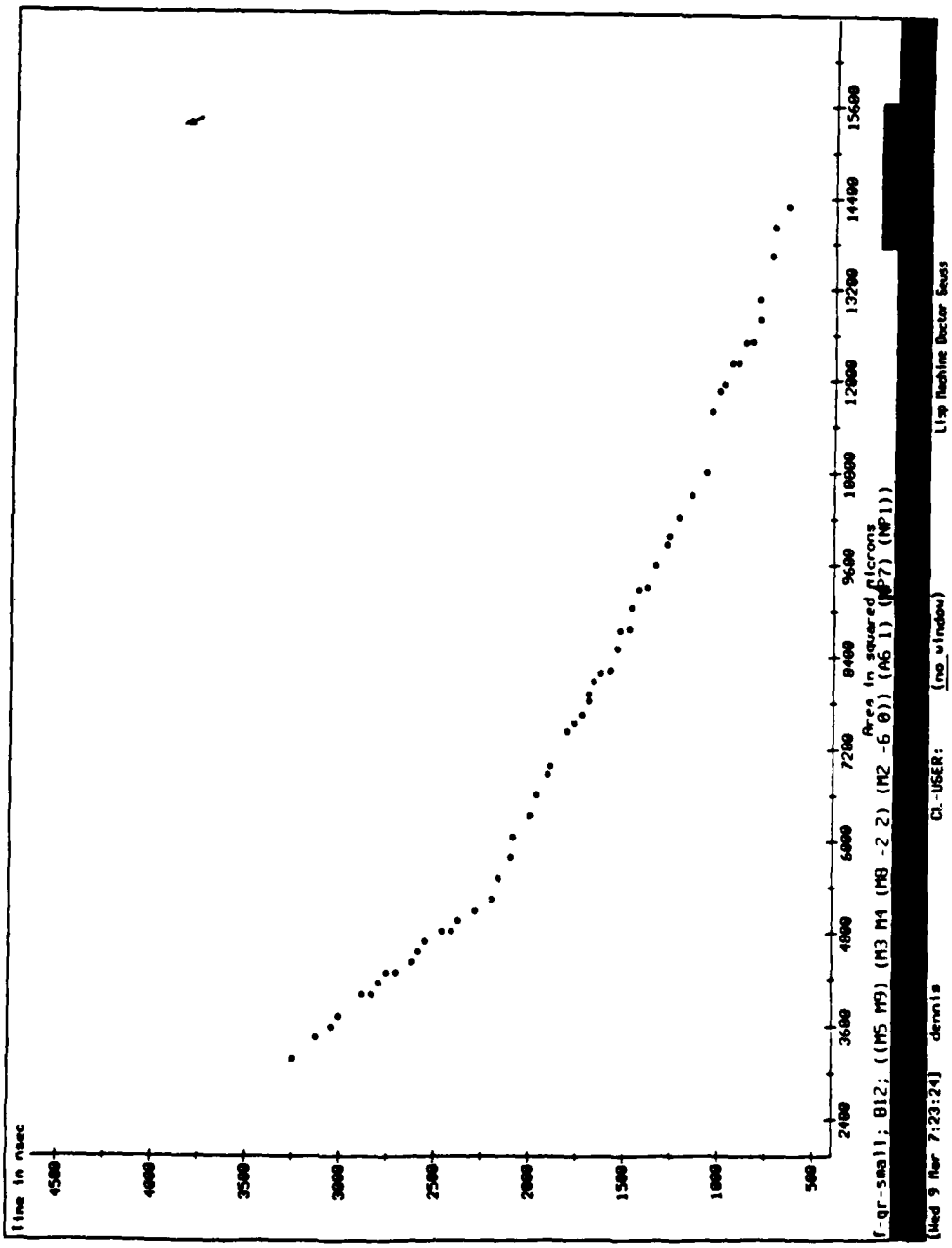

Time in nsec
4500
4000
3500
3000
2500
2000
1500
1000
500
2400
3600
4800
6000
7200
8400
9600
10800
12000
13200
14400
15600
Area in squared microns
[-qr-small: B12: ((M5 M9) (M3 M4 (M8 -2 2) (M2 -6 0)) (A6 1) (NP7) (NP1))
[Wed 9 Mar 7:23:24] dennis
CL-USER:
(no window)
Lisp Machine Doctor Bass

Figure 3:

with more complicated architectures and controlling the enumeration automatically.

3.4 The Local Optimizer and its Transforms

Although the decoupled design system produces a large number of designs, most will be sub-optimal. To compensate, the designs closest to the user's goal are sent to a second module that makes local improvements in the designs. This module is called the *local optimizer*. When the optimizer receives them, the disconnected decisions have become fixed and the architecture's efficiency can be improved by taking advantage of disconnected decision interactions. The local optimizer should be able to improve the candidate's performance along all dimensions. For example, instead of using the inefficient decoupled pipeline algorithm, it can use the optimal pipeline algorithm to not only improve speed but also the area by using fewer registers.

It is important to note that the decoupled system is critical to the optimizing module. If the decoupled designer did not preview the search space, the optimizer could easily waste its effort improving a design that is inherently inferior to some other design.

The optimizer consists of a set of transforms that watch for opportunities to do local improvements. The transforms identified so far are:

1. If both an addition and a multiplication are done in the same pipeline stage, eliminate the adder and time multiplex the multiplier's adder to do the addition.

2. If a pipeline register is connected to an operator and the operator stores its inputs (or outputs) in registers, then there exists a sequence of 2 registers. Replace the 2 registers with one that does both functions.

3. If there exists a pipeline stage with extra time (ie: its delay time is less than the longest pipeline stage) then try replacing its operators with slower implementations.

4. If there exist 2 or more adjacent adders in a stage, then replace the adders with a CSA adder tree followed by a regular adder.

5. If both addition and multiplication are done in a stage and if the multiplier is a "shift by 8 non-overlapped bits" type, then try to eliminate the adder by time multiplexing the multiplier's adder (this is a specialization of another rule; it is included because it works especially well).

6. If 2 adjacent pipeline stages have lots of extra time, try merging them.

7. Find the pipeline stage that is the bottleneck (ie: stage that has the longest delay) and replace some operator in it with a faster one. Make sure the new architecture is not already done by the decoupled designer.

The interaction between the two design modules is not limited to the decoupled designer feeding the optimizer; the local optimizer may want to influence the decoupled module too.

The emphasis of this project is decoupled design and not local optimization. Issues raised by the optimizer will not be explored in great depth. There will be no attempt to push the optimizer past the state of the art. The local optimizer will be implemented in order to create a useful design tool.

The local optimizer is typically applied only to the designs that are close to the user's preference. An objection to using only the best designs is: what if the optimizer is applied to a bad architecture and the result is better than any other architecture? That is, why should only the good architectures be optimized? This issue is not significant for two reasons. First, the objection is simply expressing one extreme of the tradeoff between design time and design quality. The more designs the user optimizes the more confidence he has in the designs presented, but this information costs execution time. If the user wants to see only highly optimized designs, then everything can be optimized. The system's philosophy is that the user should have the option of getting quick feedback on designs or producing highly optimal designs. Second, local optimization has limited effectiveness; the biggest performance improvements are made by changing the global architecture. Resources should be spent on architectural changes, not local ones. Methods to alter the architecture will be discussed in the next section.

The optimizer improves the overall efficiency at two levels of architectural abstraction. Some optimizations apply to the same level as the decoupled design. For example changing operator types or altering pipelining stages can be done viewing operators as primitive units. Unfortunately, relatively few optimizations apply at this level, so typically some part of the architecture must be expanded to more detail. Examples of this type of optimization are sharing registers between two adjacent operators or time multiplexing operators that are used inside of other operators.

The optimization transforms listed above were applied, by hand, to several dozen designs. It was somewhat surprising and disappointing that in general the optimizer could not improve performance nearly as much as the user would hope. It is important to note, however, that the architectures were not chosen randomly. Rather, they were selected because they were among the best architectures produced by the decoupled designer. It is also important to point out that the optimizer cannot move architectures around the tradeoff space nearly as effectively as the decoupled designer can produce them. This emphasizes the virtues of the decoupled design technique over more standard design methods.

3.5 Linear Programming Approach to Operator Selection

Several cases of linear programming assisting computer architecture have been reported. Of this work the most relevant has been the various versions of optimal workload scheduling

of n jobs on m processors ([Mil82, Mei81]). This subsection describes a new application of linear programming in which optimal operator assignments are found. The section ends with a comparison of the decoupled design and linear programming approaches.

A linear programming problem ([Thi79],[Ign82]) fits the form: Given $\mathbf{A}$, $\vec{b}$, and $\vec{c}$, find $\vec{x}$ such that

$$\begin{aligned} \textit{minimize}: \; z &= \vec{c} \cdot \vec{x} \\ \textit{subject to}: & \\ \mathbf{A}\vec{x} &\leq \vec{b} \\ \vec{x} &\geq \vec{0} \end{aligned}$$

The less-than-or-equal constraint can also encode greater-than-or-equal and equal constraints. Although the worst case running time is exponential, experience has shown it grows as 1.5 or 2 times the number of constraints.

Operator selection can be posed as a linear programming problem by choosing a fixed delay time for the circuit and minimizing the circuit's area. Two realtionships must be encoded. The first ensures that the circuit's delay time is bounded by the fixed delay. The circuit's delay is the maximum of the path delays. A constraint for each path can force all paths to be bounded by the fixed delay. The second relationship is between time and area for each operator. Figure 4 displays a point in a time verses area plot for each multiplier in Table 1. A similar figure exists for other operator types. Although the system is constrained to choose between discrete implementations, the linear programming formulation pretends to offer a continuous set of operator choices. Imagine that an operator can be assigned to any implementation which has performance characteristics that lie inside the convex hull of these points. The convex hull describes a convex region with linear boundaries. The boundaries can be encoded with inequality constraints. Each operator instance in the circuit has its own convex hull so it can choose its own implementation. The linear programming solution will specify the characteristics for each operator through the time and area variables. Altough each solution will lie inside the convex hull, it may not correspond to a realizable implementation. A post processing phase will have to map each operator to one of the known implementations. Actually, since the overall linear programming problem is minimizing area, solutions inside the convex hull will be dominated by those that are on the minimum area boundary. Thus, all raw solutions will lie on the boundary and will be mapped to known implementations by choosing the two closest implementations on either side.

One of the advantages of decoupled design is that it produces a spectrum of designs so the user can see the possible tradeoffs between performance dimensions. Linear programming can provide the same benefit by iterating over a range of delay times. Alternatively, a

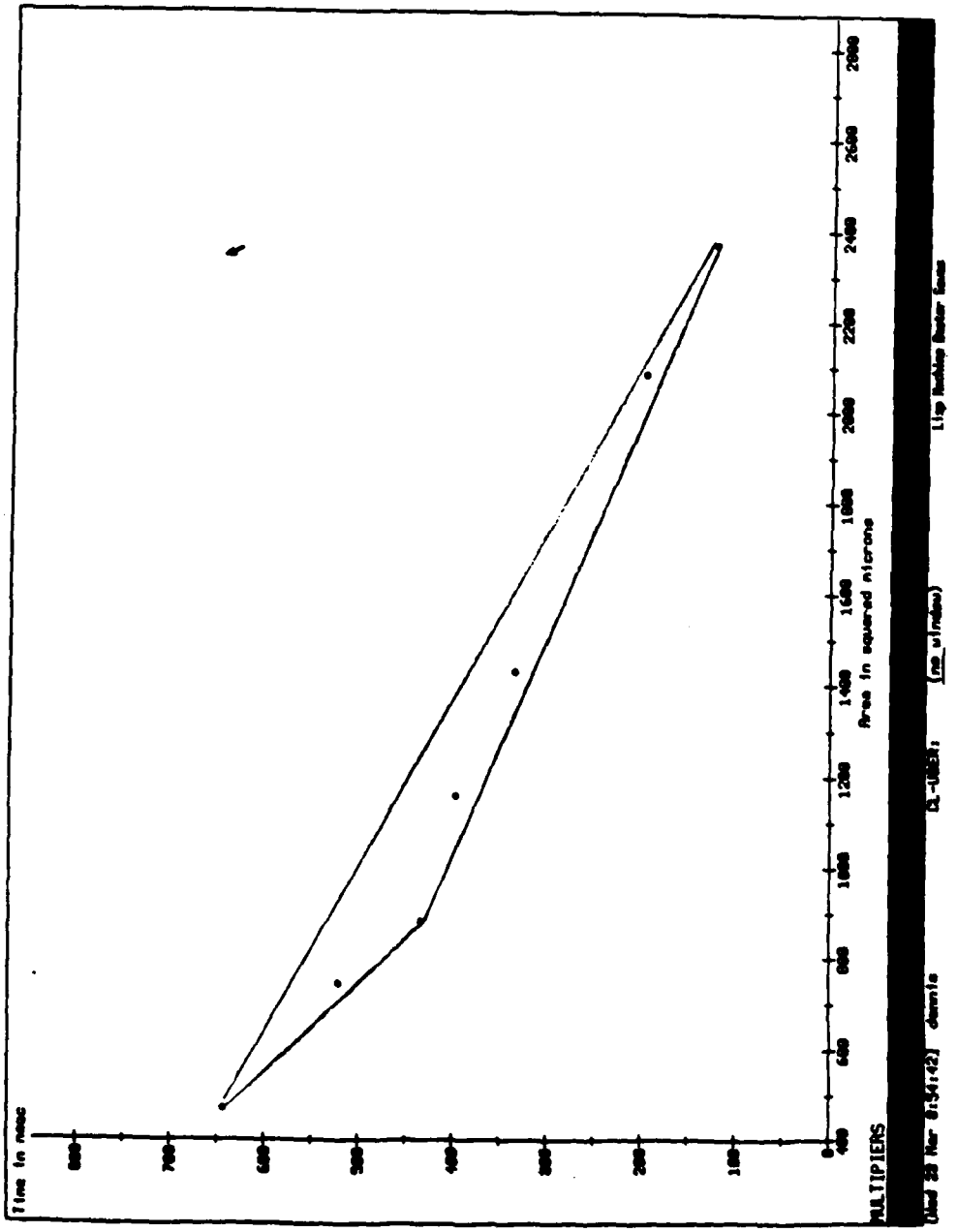


Figure 4:

more advanced use of linear programming, called parametric programming, can describe the continuous behavior of the objective function as constraint values change. It can graph a series of line segments to show how the minimum area depends on the delay time. The line segment solution still must to be mapped to known operator implementations at periodic intervals.

The basic linear programming approach has been implemented and the results for the Figure 1 dataflow graph are given in Figure 5. The open squares are the linear programming solutions and the solid diamonds are solutions that have been mapped to known operator implementations. The results took 110 seconds to be generated.

The results in Figure 6 show that the decoupled design solutions compare favorably with the linear programming solutions. The open squares are the decoupled design dominating solutions of Figure 3, the diamonds are the linear programming raw results, and the triangles are the linear programming mapped designs. In comparing the two approaches there are several factors to consider. Linear programming's primary advantage is its guarantee of optimality. The raw solution gives a lower bound on the performance tradeoff. The linear programming mapped implementations are not guaranteed but the user can see how close they are to the lower bound. The decoupled design issues become important when the system becomes more complex. When more performance factors are considered, the decoupled design style adapts easily. Linear programming, on the other hand, must iterate over $n-1$ dimensions while minimizing the n-th. Some initial thoughts on using these methods to place pipelining registers produced non-linear terms. The non-linearity requires substantial modification to the efficient linear programming scheme. Decoupled design does not rely on linearity so it applies equally well. A disadvantage of decoupled design is its long development time which cannot be transferred across domains. The developer must iteratively refine the enumeration language so it can express the best designs, and then build a system which controls the enumeration. The current preference is to use linear programming because of its optimality. This does not condemn decoupled design. On the contrary, as the design system becomes more complex the case for decoupled design will become stronger. This is especially in light of the excellent results illustrated in Figure 6. Another option is to use both methods, say, using linear programming on a few points to see how good the decoupled design solution is.

4 Architectural Modifications

A design system needs to be able to modify structures so it can produce good designs in any region of the performance tradeoff space. Different architectural structures cover different performance ranges and have different resource needs. The best structure depends on the user's needs.

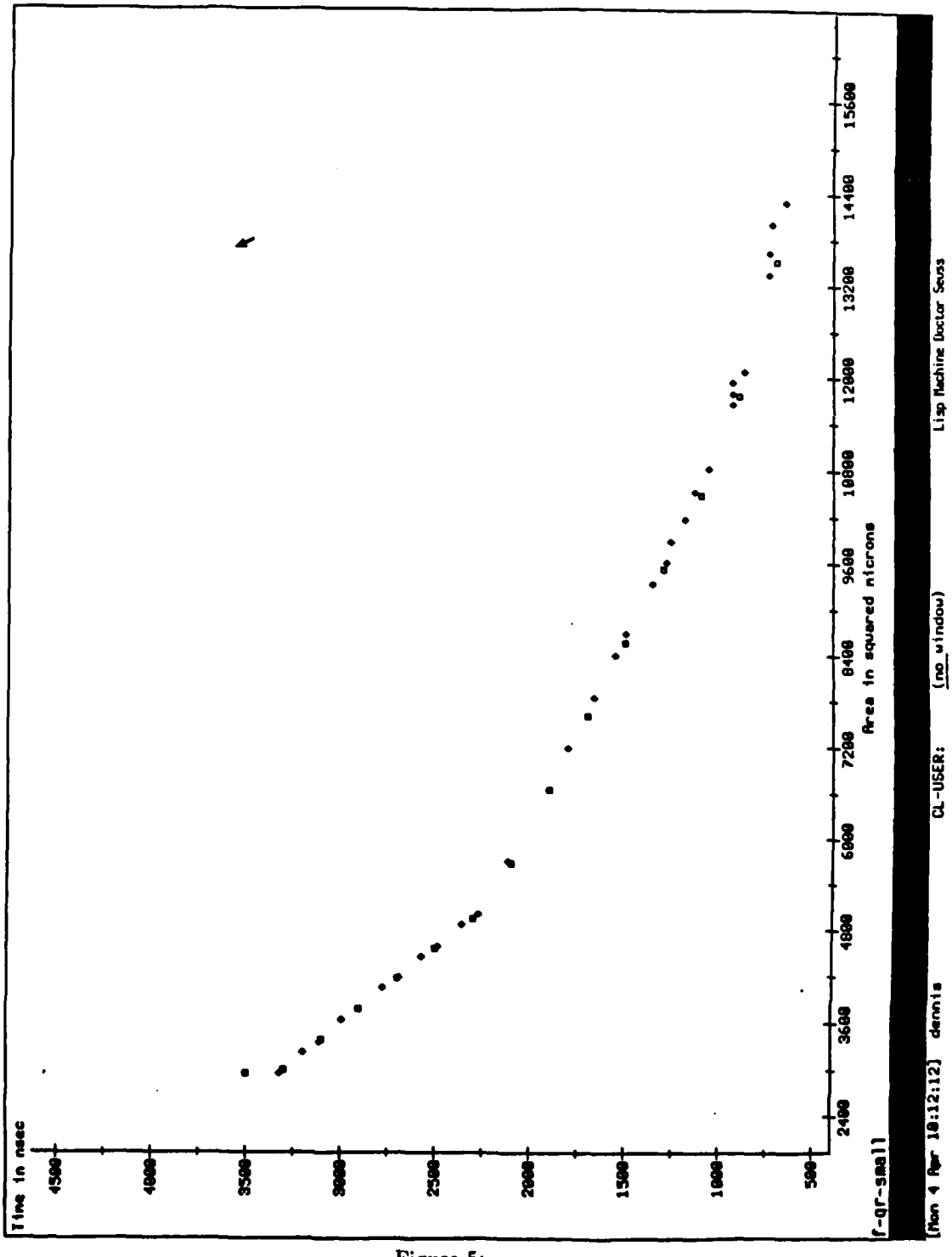

Time in nsec
4500
4000
3500
3000
2500
2000
1500
1000
500
2400
3600
4800
6000
7200
8400
9600
10800
12000
13200
14400
15600
Area in squared microns
f-gr-small
(Mon 4 Apr 10:12:12) dennis
CL-USER:
(no window)
Lisp Machine Doctor Seuss

Figure 5:

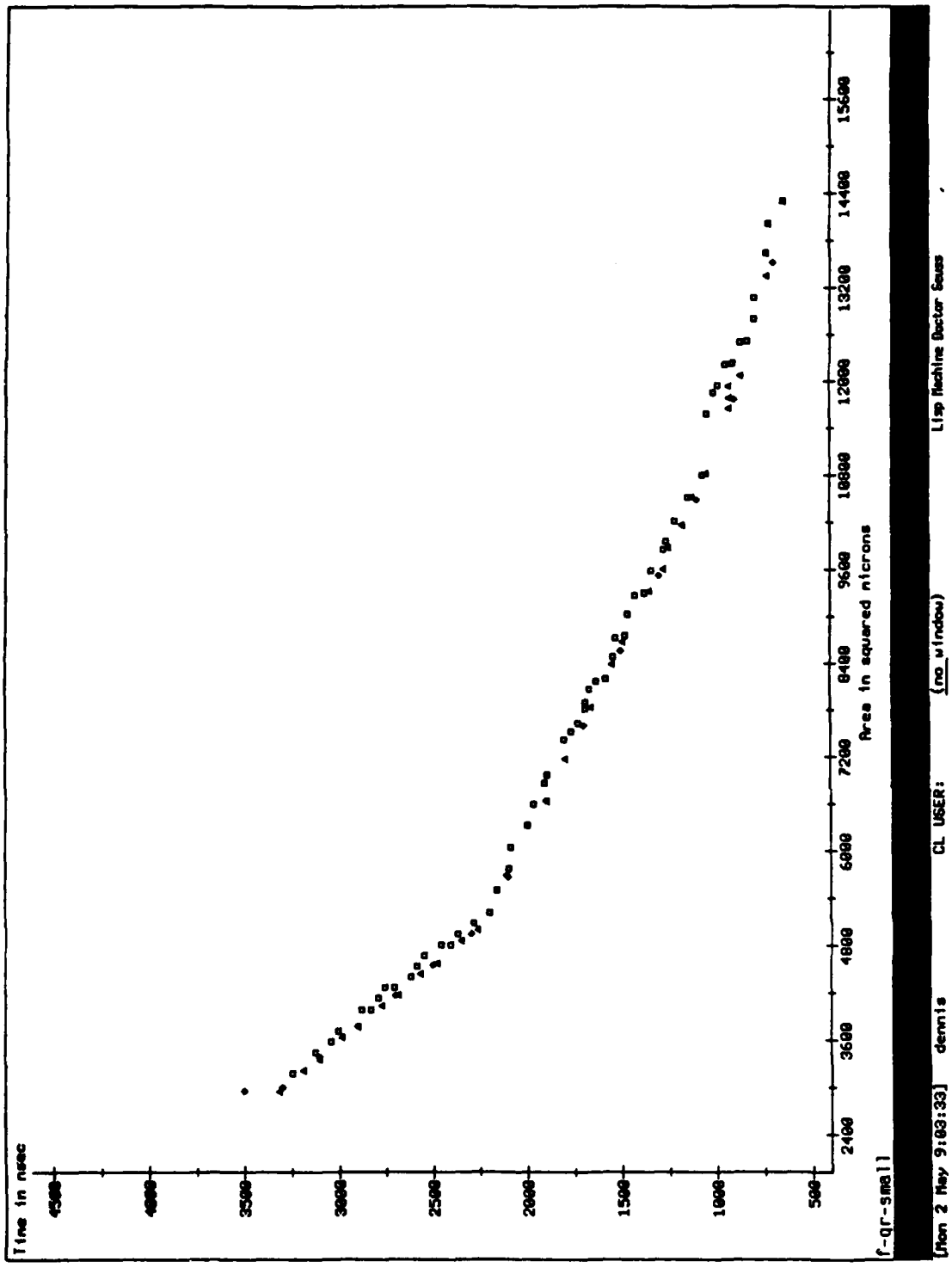

Time in nsec
4500
4000
3500
3000
2500
2000
1500
1000
500
2400
3600
4800
6000
7200
8400
9600
10800
12000
13200
14400
15600
Area in squared microns
[Mon 2 May 9:03:33] dennis
CL USER:
(no window)
Lisp Machine Doctor Seuss
f-gr-small

Figure 6:

4.1 Alteration Strategies

Alteration strategies are abstract transformations that capture the commonality among a set of more specific transformations. Since the specific tranformations may come from different domains, alteration strategies are "domain independent" in that they apply to many but not necessarily all domains. Every architectural change produced by the alteration strategies preserves the design's functionality; only the performance will change. The system can use this higher level representation in several ways. The commonality can justify a transformation to the user, allow the user to specify a transformation in a convenient way, and may allow the system to produce structural changes that were not anticipated. The alteration strategies described in this section are refinement, sharing, synchronization, and efficient constructs.

4.1.1 Refinement Strategy

Refining an object into its constituent parts can assist the design of efficient and novel architectures. The use of *refinement* in this proposal will differ a bit from the traditional approach in that it will make the distinction between 2 types of refinements: re-expression and de-structuring. In this system every designed object has an input/output (I/O) language describing its function. If the I/O language of the object is at the same level of abstraction as the I/O language of its parts, then it is a *re-expression* refinement. If the part I/O language is at a lower abstraction level, then it is a *de-structuring* refinement.

To clarify, let's give an example. The I/O languages of parts and wholes are transformations of an input stream of some data type to an output stream. In a re-expression refinement, the data types manipulated by the parts are the same as the whole's. Thus, the re-expression of the summation operation ($\sum_{i=1}^{n} a_i$) into its parts ($a_1 + a_2 + \ldots + a_n$) is a re-expression refinement since the addends in both expressions are the same data type. An obvious use of re-expression refinement is that it allows for more compact specifications, so programs or people have less input to process. In de-structuring refinement, the data types manipulated by parts are at a lower level of abstraction than those of the whole. There is a rich hierarchy of data types in signal processing. If we start with bits as the lowest data type we can combine bits into positive integers which can be used in signed integers which can be generalized to fixed point or floating point numbers. Moreover, these data types can be used in complex numbers which can be combined into vectors and further combined into matrices. So, an example of de-structuring refinement is refining the dot product operator which operates on vectors ($\vec{a} \cdot \vec{b}$) into its part definition ($\sum a_i b_i$) which is a part description that uses scalar data types. With respect to designing for different performance tradeoffs the most important difference between the 2 refinements is that de-structuring refinements often involve repetitive and regular instantiations of the part data type to create the whole's data type. The repetitive and regular nature is usually reflected

in the computational structure of the parts as they create the whole.

Since there are 2 types of refinements, there are also 2 types of *aggregations* or ways to combine a set of parts into a whole. Re-expression aggregations combine parts into wholes where both manipulate data types at the same level of abstraction. In practical terms a re-expression operator can be created when any 2 adjacent operators appear together frequently. It is very easy to create a re-expression aggregation. A common example is a multiply/accumulate structure which is detected when there is a multiplier whose output feeds an adder and the adder's other input is fed back from its output via an accumulating register. Variants of this might replace the multiplier with a vector scale operator or even a matrix multiplier.

The other type of aggregation is de-structuring where the data types of the parts and wholes differ. The advantage of composing a set of de-structuring operators is that once the higher level operator is discovered there is no reason to remain committed to that particular refinement. There may be alternative ways to refine the higher level operator. This provides additional degrees of freedom in the implementation. An example would be combining a set of scalars into a vector and doing the corresponding vector operation [2].

Refinement is not limited to operators. Both buses and memories can be refined or aggregated. For example, it is convenient to have the concept of a vector bus or a bus that moves entire vectors at a time as, say, an input to a vector operator. Vector buses can be refined into a set of buses each carrying an element. Each of these can be further decomposed into buses carrying bits.

4.1.2 Sharing Strategy

The second alteration strategy is sharing of parts. If 2 parts of the design structure are the same then it might be possible to share them. The benefit is the design has fewer physical

[2] One could criticize the need for the aggregation abilities since if the scalars could have been combined into a vector then the user would have specified them as such. But, often there is more than one way to express a computation and one way might aggregate one data type at the expense of another. To illustrate, let's look at the HH computation described in Appendix A.2. The calculation has a term: $\sum_{i=n}^{N} u_i^{(n)} a_{im}^{(n-1)} = y_m^{(n)}$ where m = n to N. This is the easiest way to represent the computation since i and m vary with n. If the index m varied over a constant range, say, from 1 to N, then the computation describes a vector scale/accumulate structure: $\sum_{i=n}^{N} u_i^{(n)} \vec{a}_i^{(n-1)} = \vec{y}^{(n)}$. A design system could get the effect of vector operators by aggregating the a_{im}'s into row vectors and padding them with zeros to get constant length vectors. Even though more operations are required, the computation could be faster if special vector scale/accumulate hardware exists. An alternative aggregation of the original computation is to view the a_{im}'s as column vectors of length $N - n$. In this case the system does not have to zero pad the vectors because they are to be dotted with the $\vec{u}$ vector whose length varies in the same way. The computational structure is a set of m parallel dot products where the length of the dotted vectors varies with n: $\vec{u}^{(n)} \cdot \vec{a}_m^{(n-1)} = y_m^{(n)}$. The dot product equation does not show the dependency of the vector size to n. Notice that the different aggregation has changed the computation structure to one that is easily parallelizable. A third alternative is to zero pad both the rows and columns of the a matrix and do a full fledged matrix times vector operation.

parts. But sharing often has a cost: typically there is extra hardware needed to implement the sharing. Also, the 2 usages of the shared part must be temporally disjoint. Sometimes 2 parts don't have to be identical to be shared. If one part subsumes the functionality of another or if they are both subsumed by a third, then the 2 parts can be replaced by one and it is still considered sharing. Functional subsumption will be discussed in subsection 4.1.4. Sharing and refinement interact synergically. If part A and part B cannot be shared, it may be possible to share sub-parts of A and B. For example, although a multiplier and an adder cannot be shared at the top level of abstraction, they can be shared if the multiplier is refined because there is an adder in the multiplier.

As an illustration of sharing interacting with refinement, let's look at the HH computation described in the footnote of the refinement subsection. The conclusion was that the inner term ($\sum_{i=n}^{N} u_i^{(n)} a_{im}^{(n-1)}$) could be implemented with either dot products or a vector scale/accumulate. The full computation has the form

$$a_{k,m}^{(n)} = a_{k,m}^{(n-1)} + b_n(\sum_{i=n}^{N} u_i^{(n)} a_{im}^{(n-1)}) u_k^{(n)}$$

where b_n is a term that depends on n and can be considered constant within the n^{th} iteration. Once the summation is computed, the computation reduces to $a_{k,m}^{(n)} = a_{k,m}^{(n-1)} + b_n \times y_m^{(n)} \times u_k^{(n)}$ Now, the system can create a vector scale/add structure by aggregating the terms by either k or m. Vector scale/add computation can be made identical to the summation's vector scale/accumulate by feeding the adder output back to the adder. Thus a reasonable design idea would be to use the vector scale/accumulate structure to compute the summation and try to share it with the vector scale/add of the rest of the computation.

Just as refinement was not limited to operators, neither is sharing. If a vector bus is refined into a set of element buses, then they can be efficiently shared as long as they are temporally disjoint. Memory can also be shared.

4.1.3 Synchronization Strategy

The third alteration strategy is synchronization of parts. *Synchronization* uses de-structuring refinements to create independent part computations that can take advantage of pipelining. Unlike pipelining, it provides computational benefits even if there is only one instance. The concept is best illustrated by an example. Given 3 n × n matrices to multiply together, the obvious way to compute it is to multiply the first 2 matrices together and then multiply the intermediate result with the third. The total time is $2n^3$ multiplies. Synchronization can cut the computation time to $n^3 + n^2$. The matrix multiply computation can be refined into individual dot product calculations. Each dot product produces an element for the result matrix in n multiplies. Instead of waiting for all n^2 elements of the first result to be computed, the computation of the second matrix multiply can begin as soon as the first

row of the intermediate result is available. Assuming the speeds of the 2 matrix multiplies is the same, when the second matrix multiply has produced the first row of the final answer the second row of the intermediate result should be ready. Thus the second multiply lags the first by one row. The total multiply time is n^3 for the first multiply plus n^2 for the last row of the intermediate matrix for a total time of $n^3 + n^2$. This example shows how synchronizing pipelines the internals to decrease the overall computation time. It is restricted to de-structuring refinements because they have regular structure. Synchronization is also restricted to operators that are adjacent in the dataflow graph. Since synchronization depends on refinement, it is sensitive to additional degrees of freedom introduced by alternative refinements.

4.1.4 Efficient Implementation Constructs Strategy

The fourth alteration strategy is efficient use of implementation constructs. The idea is to design with parts that use resources efficiently. Sometimes a construct's applicability or functionality is very specialized and thus the user may not know about it, may not know that it can be used in this design, or it may be difficult to express in the input language. Yet, if this construct is extremely efficient it should be considered in implementations. Examples of specialized constructs include: implementing multipliers and especially dividers using a ROM to take the log and anti-log and using addition or subtraction to do the equivalent logarithmic calculation; using a special adder that adds n numbers by columns instead of a row at a time so computation time is log n instead of linear; special implementation technologies like charged coupled devices (CCD) and surface acoustic wave (SAW) hardware that compute convolution as an analog process and are therefore very fast; quantized multipliers; using look-up tables to implement functions; truncating precision on operators; widening accumulators for more precision; separating summations into pieces and doing them in parallel; and doing summations in a different order.

The search for efficient constructs uses 2 mechanisms to find alternative implementations: aggregation and functional subsumption. Aggregation recognizes groups of operators as implementations of a known operator. The system can substitute alternative implementations that are more efficient. Recognizing a convolution operation and substituting a CCD is an example. A functional subsumption hierarchy can also find new implementations. Operator A functionally subsumes operator B if A's function is a superset of B's. Functional subsumption adds more options to the list of possible implementations of an operator. For example, a two's complement multiplier functionally subsumes a negation operator, so it could be used instead.

Another relevant distinction is between the specified function and the desired function. The specified function is the user's functional specification; the desired function is the

user's intent which is inaccessible to the program [3]. The specified function is only an approximation to the desired function, so a design that relaxes the specification may still meet the user's needs. Often, functional subsumption improves efficiency by substituting an approximating operator for the specified one [4].

4.2 Alteration Strategies in Design

4.2.1 Advantages

Design could be done with a set of production rules or in some programming language. Rules have the problem of conflict resolution. Programming languages have the problem of having to specify exactly the order of operations. The steps in the design process often depend on the input so a more flexible method of applying design knowledge is preferred.

Alteration strategies encode higher level knowledge which may be useful in design. The potential benefits and uses described in this section constitute various directions that will be explored in using alteration strategies. Some advantages are more certain than others.

Alteration strategies can be thought of as sets of transformational paradigm transforms. The transforms are grouped according to domain independent features, and each alteration strategy has properties that are common to the transforms in the set. The properties can be used to eliminate from consideration an entire set of transforms. For example, no sharing transform can be done unless there exist 2 objects that are sharable.

A Strategy can be interpreted as a justification for a transform. A justification can be useful for several reasons. First, it provides an explanation to the user when he inquires about what the transform is trying to achieve. Second, it can assist in transform conflict resolution. A basic operation of a rule based systems is deciding which transform to apply. Each transform has an antecedent (the "if" part). These conditions are necessary but not sufficient for applying a rule. If several rules apply, conflict resolution must choose between them. A simple scheme is to prefer antecedents with more terms. This ad hoc method is based on the syntax of the transform instead of its semantics. Another possibility is to treat conflict resolution, itself, as a problem to be solved using a set of meta-rules ([Dav80, LNR87]). Although meta-rules provide a mechanism for solving conflict resolution, they provide little help about using the mechanism for multiple performance criteria design. Explicit justifications allow the system to treat the qualifying rules as more than arbitrary

[3] This distinction is similar to that between specification and behavior described in [MSKC+83]

[4] A new alteration strategy organization is emerging: refine/aggregate using a library of known implementations, share/replicate resources, temporal parallelism by interleaving computational instances, and functional subsumption. These changes to the architecture do not necessarily correspond to similar changes for the data manipulation. For example, an architecture that processes data serially can still be shared, and replicating a serial data architecture may not result in a parallel data architecture (it could be used for interleaving).

transforms. The justification can predict the effect of the transform without applying it. Conflict resolution chooses the transform with the best predicted outcome. The decision could be as simple as choosing a transform that affects the critical path over one that changes a peripheral part of the design or preventing a subgoal from being clobbered. For example, assume there are 2 applicable transforms: one that wants to share pieces of a regular structure and another that gets rid of registers between operators to save both time and space. Adding "dummy" registers between operators can be useful at times. In particular, adding registers between operators in a regular structure (like a multiplier) creates a structure where each piece is identical and isolated from other pieces. Identical pieces can be multiplexed into one piece. This could provide a much bigger savings in space than shaving a few registers. This would be especially important if the system needed to reduce the area dimension.

An exciting direction to take alteration strategies is to emphasis their ability to separate the transformation operation from the transformed object. The strategies are terms in a language to describe change without referring to the objects being changed. When a transformation is needed, the strategy and object are combined to select an appropriate transform. The object can be specified by name or by object decription which is executed at transform-time to determine the object to be changed. For example, alteration strategies and object descriptions can be used to make statements like "share all the memories in the system" or "share the matrix memories in the system" or even "share the 'expensive' memories in the system." Another example would be "replicate all resources that are both inexpensive and are involved in the bottleneck." Alteration strategies support a language that can describe a range of design processes from extremely specific algorithms to a loose sketch of what to do next. There are, of course, issues to resolve in directing the search in the less directed plans; one idea is presented in section 5.

Alteration strategies can also bridge the gap between the user's intuition and a particular transform. Strategies are domain independent so the same strategy vocabulary can be used in many domains. In fact the same strategy term might have different meanings when applied to different sub-domains of the same domain. In VLSI, sharing buses and sharing memories imply 2 very different sets of transforms. In general, using alteration strategies as generic transformation terms allows the user to refer to different set of transforms via the commonality between them (ie: the domain independent aspects). This has the advantage of hiding the details of how the transforms change structure and greatly reducing the user's communication burden.

Finally, the commonality encoded in alteration strategies can be used as more than mere summaries of domain specific transforms. Since the system will be constantly modifying architectures, it is plausible that it will create a structure that was not anticipated by the author of the transforms. Thus, none of the domain specific transforms will apply.

But, since alteration strategies encapsulate generic transformations, it would make sense to apply one of them. The strategy should be enough to create a new structure even though no domain specific transform applied.

4.2.2 Discussion

Although the emphasis in this section has been on domain independent alteration strategies, it should not imply that this design system has no need for domain specific transformations. *Simplification transforms* are domain specific transforms that "tidy up" the design after the alteration strategies have been applied. Simplification transforms are used to catch design violations that are considered inappropriate no matter what the intended design is. A common violation is a design fragment that is clearly inefficient. One of the requirements for a design structure to be finalized (ready to be sent to the decoupled design module) is that all simplification transforms have been tried but none can be applied. A simplification transforms example is given in Appendix C.

How does the system choose among the alteration strategies in order to reach the user's performance preferences? If the design's performance is an order of magnitude away from the desired levels, then the transforms that reduce that difference should be applied. When the design performance is closer to the user's target the decoupled design process make the transform choice more difficult. For example, given that the system is constrained to use the direct map architecture class, if the user wanted to make it faster the system could pipeline it or if the user wanted to make it smaller the system could share it. But, what if the system pipelined it and used slower operators or shared it and used faster operators? The effect on both time and area are unknown. The designer could try to predict the effect but that would be hard and unnecessary since it is easier to try the combinations. But then the problem becomes lack of knowledge about which transform to apply for a particular effect. There is no longer a strong correlation between goal and transform, so transforms are less effective for targeting a design to a particular performance goal. An alternative is opportunity-oriented transform application. If a transform results in a design that uses its resources more efficiently then the transformed design is expected to have a better performance verses resource tradeoff. The specific time or area values will be unknown, but the tradeoff between the two will be better. This assumes, of course, that the range of designs will fall somewhere near the user's goals.

4.3 Examples

This subsection will illustrate the alteration strategies creating different architectures from the same dataflow graph. Specifically, the example will describe how the alteration strategies map a dataflow graph into a standard 3 bus architecture and then describe how permutations of the mapping can create different designs. The algorithm is the CG computation given

in Appendix A.4 and shown in Figure 17. It should be emphasized that the system is not expected to find the following transformation sequences without domain specific guidance.

The first step is to examine the dataflow graph for leverage points and opportunities. Leverage points are structures that will have a large effect on the entire design if they are changed. Typically, they are structures that consume lots of resources or are bottlenecks to performance. In the CG dataflow graph, the main leverage point is the matrix-times-vector computation because it is the only operator that has computation time worse than linear (in the size of the input vector). Other possible leverage points are the linear time operators: dot product, vector scale, and vector addition/subtraction. Opportunities are typically particular combinations of operators that can be taken advantage of. For example, the CG graph has 3 instances of vector scale followed by vector add or subtract. Moreover, there are no instances of the 2 operators outside of those 3 combinations. This suggests building a fast operator to do both operations (ie: a re-expression aggregation). But if there is a fast operator to do vector scale/add then refinement and sharing should recognize that it would be advantageous to use it to implement the matrix-times-vector operator. Actually, the matrix-times-vector operator uses a vector scale/accumulate operator but an accumulator can be implemented with an adder. There may be some conflict. The vector scale/accumulate might want to increase the throughput of the scaling operation with pipelining, whereas the vector scale/add has only one input instance to scale. It would want to take advantage of interaction between scaling and adding by sharing the adder.

This analysis requires sophisticated deduction. A pattern finding program would have to notice that all instances of the vector scale operator are followed by a vector add or subtract. It would suggest this as a re-expression aggregation. The aggregation process should recognize this pair as a vector scale/add operator, and it should annotate the dataflow graph with this information. Once the vector operator combination is in the dataflow description it should be easy for the sharing alteration strategy to take advantage of the opportunity.

The second step is to take advantage of the opportunities and leverage points while transforming the design into a 3 bus architecture. A vector scale/add/accumulate operator is built and it is used for both the matrix-times-vector operator and the vector scale/add operators. They will all share one physical operator. The only remaining operators are dot products and divisions. Since a serial architecture is being built, the remaining operator instances should be shared into one dot product operator and one division operator. Each physical operator is buffered by multiplexors at its inputs and outputs so it can be shared. The transformations so far have changed the dataflow graph of Figure 17 into Figure 7. An associated timing description must also be produced. This architecture should not be implemented, because its irregular interconnection makes it inefficient. It can be transformed into an intermediate representation that is more regular by changing the multiplexing units.

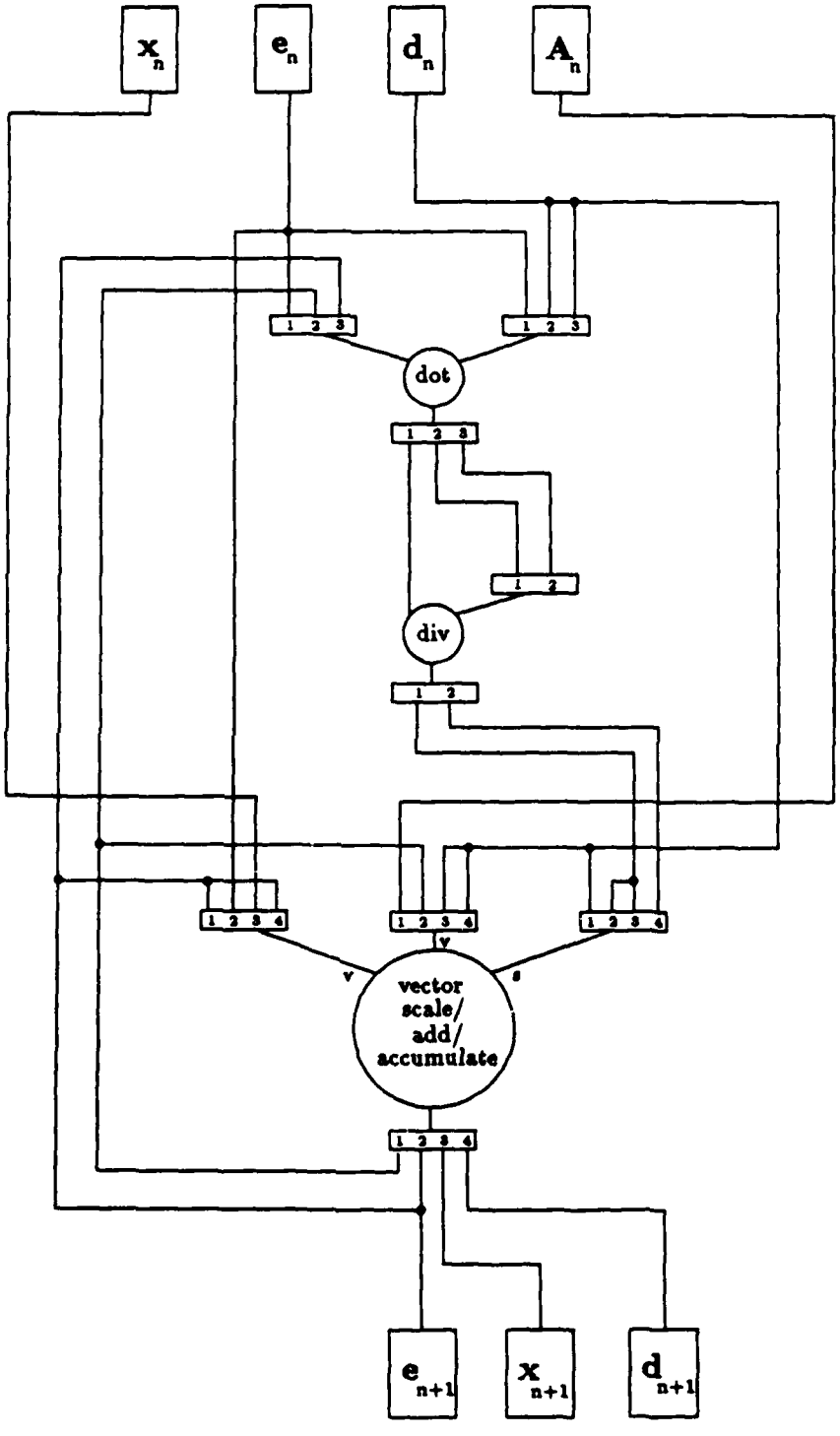

x n
e n
d n
A n
1 2 3
1 2 3
dot
1 2 3
1 2
div
1 2
1 2 3 4
1 2 3 4
1 2 3 4
v
v
s
vector
scale/
add/
accumulate
1 2 3 4
e n+1
x n+1
d n+1

Figure 7:

First registers are inserted on the wires between every operator. Then all the multiplexors are replaced with buses. This is shown in Figure 8. After creating this architecture, some daemon might notice (use the same mechanism that looks for opportunities and leverage points) that all the inputs to the division operator come form the dot product operator. Thus, the 2 operators can be combined into a re-expression aggregation. It is not enough to look once and find all possible opportunities, they also arise as the design evolves. The 2 operators can be combined by building a "mini" 3 bus architecture. The 2 operators are accompanied by 3 registers to hold values as shown in Figure 9a. Figure 9b replaces one bus with a multiplexor. This is the operator that will be used in the larger architecture. To complete the transformation to a 3 bus architecture, all the registers that held intermediate values in Figure 8 are combined into one large memory. The result is shown in Figure 10. It has 2 specialized operators, 3 input buses, an output bus, and a large memory. This structure is given to the decoupled design module which will try different implementations of the primitive operators.

One problem with this architecture is the 3 input buses force the design to use a 3-port memory. Three port memories are expensive. One remedy is to supply the input buses from separate memories. That is, one of the known refinements of n-port memories is to create n different memories where each memory contains only the variables that it needs. The output bus is extended to feed all memories. Figure 11 shows the single large memory split into 3 memories.

An alternative solution to the 3 port memory problem is to implement it with a less expensive memory. The easiest and most obvious way is to use a 1 port memory to simulate a 3 port memory. This design is straight forward and should be done by the system. It turns out that a very elegant solution to the 3 port memory problem is found serendipitously when the system solves a related problem. Buses occupy large swaths of area. Two of the three buses of Figure 10 are significantly more efficient than the third, because they transport a complete vector for every scaler the third transports. Therefore, it might be worthwhile to share the scaler bus with one of the vector buses. Figure 12a shows a simplified architecture with 2 of the 3 buses shared. It is plausible that the system will stop here and not produce any more optimizations. If, however, this part of the architecture is a dominant part then the system could explore the design space more thoroughly. If the system refines the implementation of the 3 port memory into 3 single port memories as is done in Figure 12b, then another inefficiency is uncovered. Two single port memories inside the 3 port memory operate in parallel only to be serialized by the multiplexor; a single memory could perform the same function. This violates an simplification transform (see Appendix C). So the 2 one port memories, the 2 buffer registers, and the multiplexor are replaced by a single 1 port memory. Figure 12c shows the result. Notice the system has eliminated the expensive 3 port memory. Another simplification transform would notice that Figure 12c's architecture can

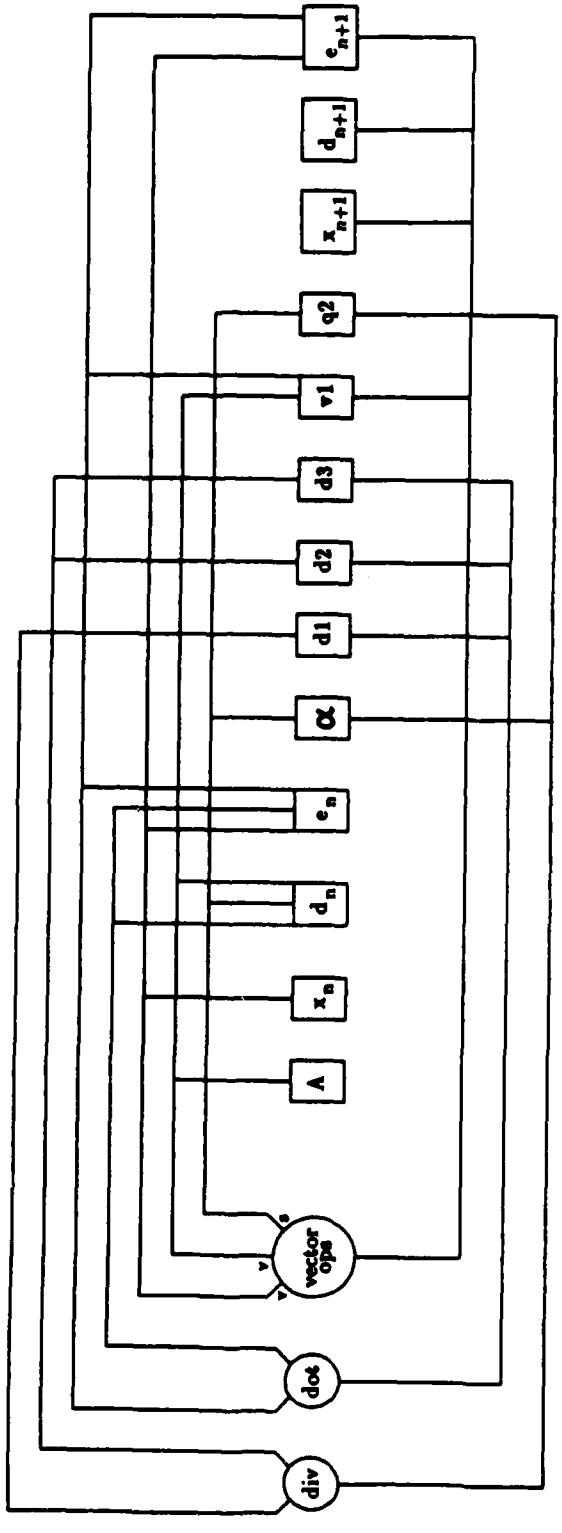
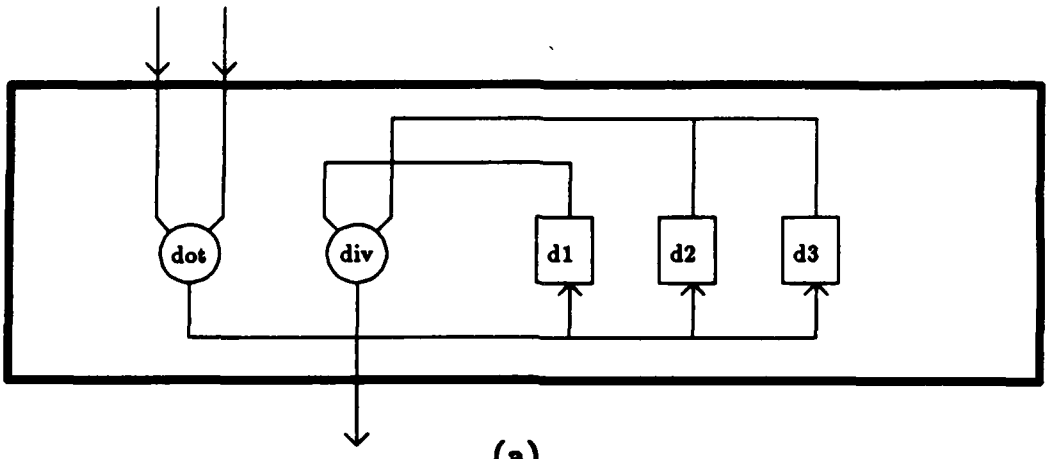
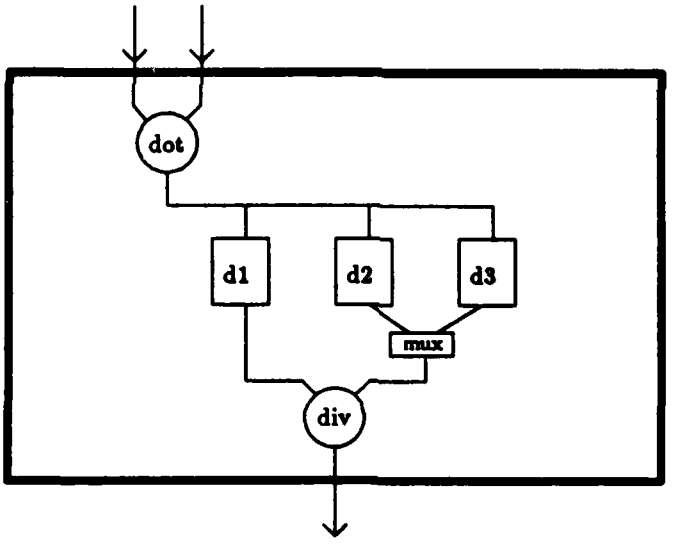


Figure 8:



(a)



(b)

Figure 9:

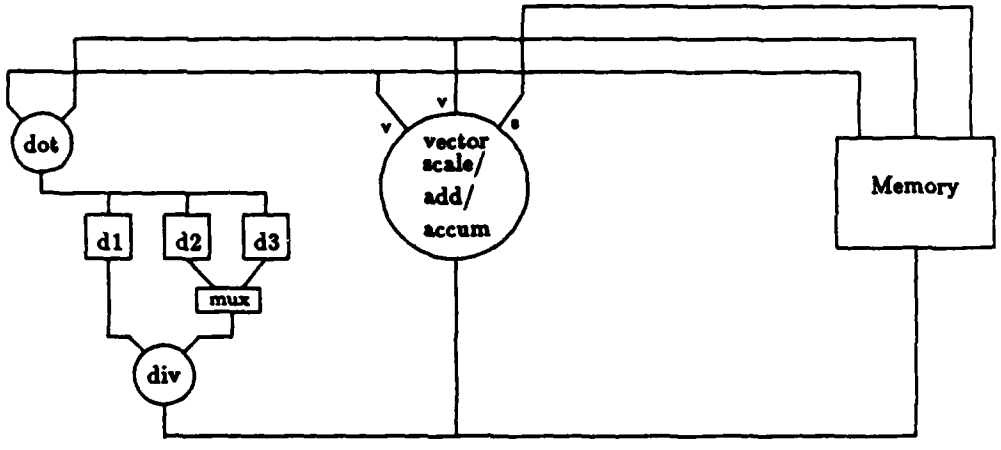


Figure 10:

be further optimized by eliminating one of the registers after the demultiplexor. The values that were buffered by that register could be sent directly from the memory. Removing the vector register saves more time. The fully instantiated architecture is shown in Figure 13.

There are other ideas that can be explored for the CG computation:

- the dot product and the matrix-times-vector could be shared leaving the vector scale/add by itself.
- the division operator is slow because it is iterative; try making the dot/division operator a separate processor.
- all operators could be shared into one processor whose structure is customized for them.
- split the dataflow graph into parts that can be done in parallel and implement it with multiple processors. Each processor can be customized for its datapath.

To summarize briefly, this example has shown how the alteration strategies can be used to transform a dataflow description into several versions of a serial architecture. The process is not a rigid step-by-step procedure since opportunities may appear as the design progresses.

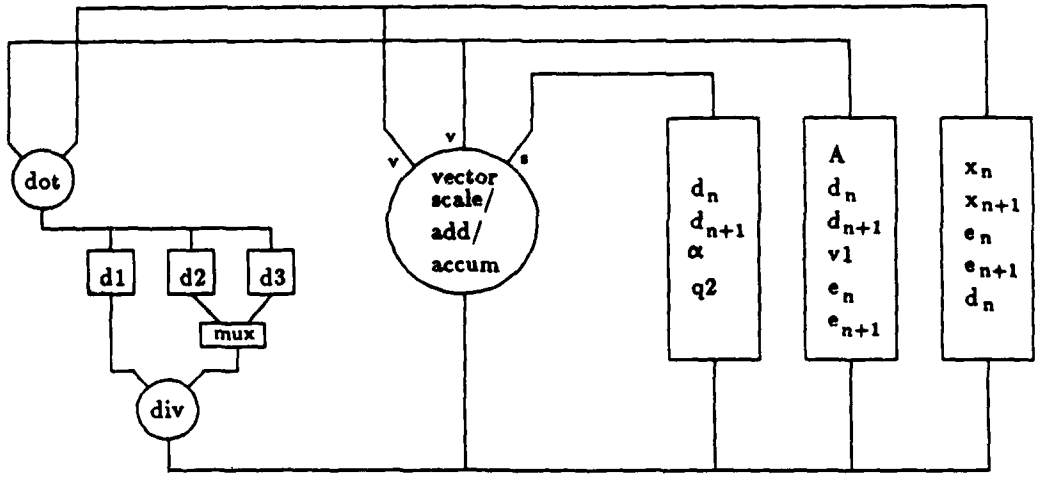

dot
d1
d2
d3
mux
div
v
v
s
vector
scale/
add/
accum
d_n
d_{n+1}
α
q2
A
d_n
d_{n+1}
v1
e_n
e_{n+1}
x_n
x_{n+1}
e_n
e_{n+1}
d_n

Figure 11:

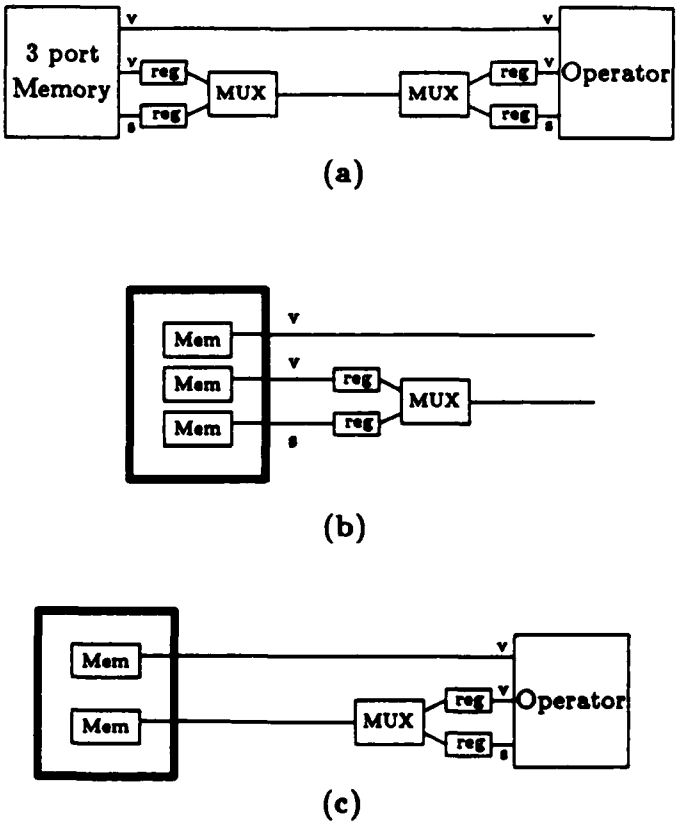

3 port
Memory
v
reg
MUX
MUX
reg
Operator
s
(a)
Mem
Mem
Mem
(b)
Mem
Mem
MUX
Operator
(c)

Figure 12:

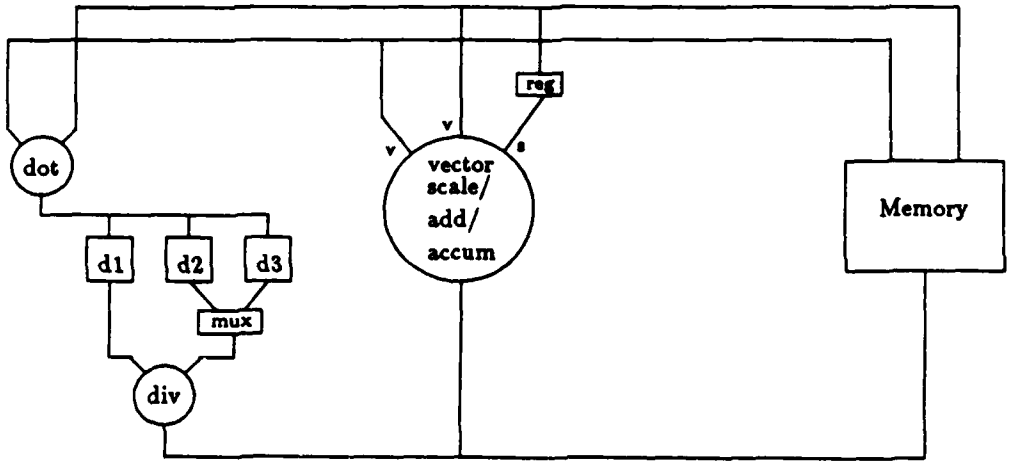


Figure 13:

The same process should work for other input dataflow graphs as well as for other types of architectures.

In conclusion, the essential ability provided by alteration strategies is to change a design's structure. Other researchers have used the transformational paradigm to accomplish this ([Kow85, Bal81]), so there is little doubt that structural alterations can be achieved by some mechanism. The emphasis in alteration strategies is that a domain independent description of the transform may be useful in other ways.

5 Controlling Search

The previous sections have described mechanisms to support designing for multiple performance goals. Decoupled design casts the design process into independent decisions, makes partial design choices to reduce the search space, and then enumerates the designs. Since there may be several ways to combine the disconnected designs, the procedure is not fully specified. Alteration strategies represent methods to change the structure of the design. The alteration is represented abstractly, so there may be several ways to instantiate it for the current design. When several alteration strategies apply, some mechanism must select one. A common theme among these techniques is they describe ways to create designs, but

they do not specify how to control their application.

This section describes a search control method called *sample search*. The idea is that only a fraction of the paths are sampled at each search node. The desirability of searching that node further is judged by the samples returned.

5.1 Motivation

The design process can be cast in a search framework by viewing the input specification as the start node, the best completed design as the goal node, and design transformations as arcs. Each input specification defines a search space which is explored with either blind or heuristic methods. Blind search systematically expands nodes and orders the alternatives. The search methods include depth first, breadth first, and bi-directional search. Although some of these methods apply to the search needed in this system, none take advantage of the heuristics available, so they exhaustively search the space.

Heuristic search methods include best first, A* search, branch and bound, beam search ([New78]), and "bounded look-ahead plus partial backtrack" techniques [5]. They use information from the domain to guide and prune search. The information is integrated into the search procedure with an evaluation function that estimates the "promise" of a node. One node's promise is compared to another's and the one with the greater potential is selected for further exploration.

Traditional heuristic search will not be effective for this design system, because applying the evalution function to non-leaf nodes is unreliable. Each non-leaf node is a partial design. Partial designs cannot be evaluated accurately. Bounds for partial designs could be obtained, but they will be fairly loose. Decoupled design forces operator instantiation to be done last, so all partial designs use abstract operators. As a consequence, most performance bounds on partial designs must be parameterized by operator instance. The operator instances are intentionally selected to produce large performance variances among instantiations which is desirable for decoupled design but undesirable for obtaining performance bounds. Thus, when decoupled design is used performance bounds will not be tight and the bounds will not prune the space well. Another impediment to accurate evaluation is that some non-leaf nodes are too uncommitted to provide a basis for reasonable bounds. The application of an alteration strategy can, for example, completely change the structure of an architecture. Precise bounds are difficult to get on a changing architecture. These factors make the evaluation of non-leaf nodes an ineffective foundation for pruning the search

[5] Pearl ([Pea84]) describes a general class of techniques called "bounded look-ahead plus partial backtrack" search strategies. The example given by Pearl is a search method that does a depth-limited search and then evaluates the current nodes. If the search is "doing well" so far, then it continues for another round of depth-limited search. If it is not, then the branch is pruned and the search backtracks to a previous node. In general, these methods can find near-optimal solutions with high probability.

space.

Even if the performance of non-leaf nodes could be obtained, they cannot be compared. The unreliable performance specification problem dictates that the system cannot tradeoff heterogeneous performance factors to find the best solution unless a design dominates in every dimension. Dominance between partial designs is not expected to be common, because the transition from one non-leaf node to another will typically slide the design's performance along a tradeoff curve instead of dropping it to an improved curve.

5.2 Generators

A *generator* is a mechanism to integrate the system's heterogeneous decision mechanisms into a uniform search structure. Any part of the program that alters the design can be considered a generator. Each generator defines a portion of the search space by offering alternative ways to change the design. Conceptually, there is a separate generator for each search node and the complete search space is a tree [6] of generators. Generators differ from simple enumeration in that a generator uses feedback to decide what to generate next. Specifically, it outputs a solution, tests the result, and then bases its next output on the test result. This is like the generate and test paradigm where the tester is fed back to the generator. The method for adapting the next output based on the previous output's result is knowledge based and generator specific.

There are several different types of generators in this design system. The first generator parses the input equation in different ways. There are many parses especially when the rules for parenthesization are not strict. The issue of whether to collect common subexpressions also changes the parse. Another generator is needed if a multiple processor architecture is used. The system has to decide how many processors to use and how to split up the dataflow graph for the processors. There is no optimal method for these decisions so the generator searches for good combinations. A generator is also used when selection of alteration strategies is done. There are often many strategy combinations possible so the generator tries to choose the best. In decoupled design if the search space is too big then the system has to re-structure it. There are many possible ways to re-structure and some are better than other. A generator should oversee the selection.

5.3 Sampling Methods

Sampling allows the system to investigate a decision without committing itself to it. By scanning a fraction of the subtree the system can discover information about the node without searching the full subtree. In addition, heuristic information is often available. Nodes in the search space are grouped into different types, and the heuristics for each node

[6] The search space is actually a directed graph, but generators represent it as a tree.

type make suggestions about which alternative to take. When the system scans a subtree, it uses the heuristics to guide the sample to the best solutions first. If the sample results are favorable then a more complete search can be initiated. Although the denser search cannot re-use the work that was done by the sample, the hypothesis is that sampling gathers information and the savings from sampling will be more than enough to make up for the sample time.

5.3.1 Generator Size Parameter

Generators can implement sampling with the generator size parameter. All generators have an input parameter that describes the size of the space that it should produce. Likewise, for each output produced by the generator a size parameter is included as input to the next level generator. Tentative semantics for the parameter are that a generator cannot produce more designs than its size parameter. The parameter is like a budget where the resource is the number of designs. The generator allocates "funds" to the most promising alternatives which are, themselves, generators. The resource is successively divided with each generator level until either the last generator is reached or until all generators receive a size of 1. When a generator is restricted to a single design, the search forms a thread of nodes from the generator to a leaf node. Sometimes a generator can deduce that it cannot meet the user's goal, so it returns its unused allocation to the parent. Since generators wait for the results of one alternative before exploring another, the extra designs can be easily redistributed to other alternatives [7].

In this domain, there are decision nodes that have a fundamentally different character from a decision node with a disjoint set of alternatives. When designing a structure there are often several simultaneous changes that can be made. Although the application of these opportunities could be serialized, it may be advantageous to represent their simultaneous application in a single generator. These *powerset generators* have a set of opportunities that can be applied to the design in any combination. The decision node alternatives are the powerset of the input opportunity set.

Another aspect to explore is whether the analysis of a previous generator can be re-used to benefit the current one. For example, if an alteration strategy transforms an old architecture into a new one, a reasonable action would be to re-use the old architecture's decoupled design analysis. There is no guarantee that the analysis applies to the new architecture or that it reduces the space sufficiently, but when generating lots of solutions often the current solution is very similar to the previous one. The number of solutions and generators grows exponentially, so speeding up the generator analysis could be very important. Implementing the re-use should be straight-forward. If there is only one generator for each node type,

[7] An alternative semantics for the size parameter would be to use Kant's idea of setting real-time deadlines for generators to meet. This may be explored if the design cardinality interpretation runs into problems.

then the generator could save the analysis as part of its state and try it when the generator is called again.

5.4 Sample Search

Sample search uses the methods in the previous subsection to heuristically search a space. The key assumption is that samples of a search space can ascertain the utility of searching that space. In particular, if a space has good solutions then some of them will appear in the sample, and if the sample does not discover any good solutions then none probably exist. The extent to which this assumption is true determines the effectiveness of sample search. The justification for the assumptions is that local heuristics exist at each decision point to direct the sampling to the good solutions first. An alternative justification is that oftentimes one of the node's alternatives will produce more efficient designs than the others. By sampling bits of each alternative, sample search can find the alternative which has the more efficient solutions clumped under it. Both of these justifications seems reasonable for the niche of signal processing architectures addressed in this project.

Sample search involves more than just cutting the search space size through sampling; the samples can be used in other ways:

- Reason about how big the search space *should* be. It may be desirable to make the space very small because the generator does not affect the outcomes much. If the generator's ability to move the design's performance is slight, don't bother searching a large space.

- In a similar vein, if there is some reasonable heuristic or guarantee that can produce a partial ordering of possible outputs, then the system should try the extremes. If an extreme trial cannot produce a solution near the user's goal then this search path is a deadend. For example, if the design is significantly oversized, the generator should create an output which includes all possible alteration strategies that reduce the design size. If it cannot alter the design's size sufficiently, then there is no use trying other alteration strategy combinations. This heuristic and the previous one show that generators don't simply shrink the design space by reducing the exponential growth factor; both heuristics can dramatically reduce the search space by planning experiments that eliminate the entire space.

- When the generator notices that a particular output has produced an exceptional result, it can take advantage of the good designs by either using a denser empirical search or using analytical methods to determine why the design was a success.

- Sampling can be used to estimate the size of the space. This information helps the generators to decide how much to increase the sampling next time.

- Generators can empirically check the effectiveness of heuristics. It would be useful to identify which heuristics were working well for this particular problem. As the program gets more experience, it should emphasize the heuristics that are producing the best results.

The theme in sample search is it may be better to find good designs empirically (using sample search) than to deduce them. Empirical methods use experiments to both prune search paths and capitalize on synergistic effects between opportunities. Deductive pruning (constraints, TMS) relies contradiction which is ill-defined for multiple criteria search. Deductive techniques to take advantage of synergies have been rare (maybe blackboard model or least commitment). Moreover, the empirical approach does not simply abandon the deductive approach's strength, namely, reasoning about an object's structure. Reasoning about structure is used after a good structure has been found, not for finding good structures in the first place. The hypothesis is that analyzing synergistic structures and using the analysis to generate more synergistic designs is more efficient than creating them through deduction.

6 Conclusion

This project investigates the synergistic interaction between human designer and computer. The particular issue explored is designing under multiple performance criteria. Trading off different criteria is a troublesome but necessary part of finding good designs. The solution is to encourage interaction between user and computer. To take advantage of its new role, the system uses three techniques: decoupled design, alteration strategies, and sample search. Although the system operates in the domain of application specific, non-regular, signal processing architectures, the ideas may apply to other areas of design.

A Signal Processing Examples

A.1 QR-algorithm for the Eigenvalue Problem

An upper Hessenberg matrix has zeros below the main diagonal except for the first sub-diagonal. Finding the eigenvalues of an upper Hessenberg matrix using the QR-algorithm involves the key transformation described in [Sch84, pages 91-95]. Given an $n \times n$ Hessenberg matrix $\mathbf{C_0}$, find $\mathbf{C_{n-1}}$ where:

For $r = 0, \ldots, n-2$:

$$\mathbf{C_{r+1}} = \mathbf{P_r^T C_r P_r}$$

where for $r = 1, \ldots, n-2$:

$$\begin{aligned}
\mathbf{P_r} &= \mathbf{I} - \frac{2\mathbf{y_r y_r^T}}{\|\mathbf{y_r}\|^2} \\
\mathbf{y_r^T} &= (\underbrace{0, \ldots, 0}_{r}, 1, s_r, t_r, 0, \ldots, 0) \\
s_r &= c^{(r)}_{r+1,r-1} / (c^{(r)}_{r,r-1} \pm S_r) \\
t_r &= c^{(r)}_{r+2,r-1} / (c^{(r)}_{r,r-1} \pm S_r) \\
S_r &= \sqrt{(c^{(r)}_{r,r-1})^2 + (c^{(r)}_{r+1,r-1})^2 + (c^{(r)}_{r+2,r-1})^2}
\end{aligned}$$

s_0, t_0, and S_0 are computed from $\mathbf{C_0}$. The sign ambiguity is resolved by the constraint:

$$|c^{(r)}_{r,r-1} \pm S_r| = |c^{(r)}_{r,r-1}| + S_r$$

The dataflow graphs for computing $\mathbf{y_r}$ and $\mathbf{C_{r+1}}$ are shown in Figures 14 and 15. A parallel version of this algorithm is given by:

$$\mathbf{C_{r+1}} = \mathbf{C_r} - \mathbf{v_r p_r^T} - (\mathbf{q_r} - \alpha_r \mathbf{y_r}) \mathbf{v_r^T}$$

where

$$\begin{aligned}
\mathbf{p_r^T} &= \mathbf{y_r^T C_r} \\
\mathbf{q_r} &= \mathbf{C_r y_r} \\
\mathbf{v_r} &= 2\mathbf{y_r} / \|\mathbf{y_r}\|^2 \\
\alpha_r &= \mathbf{p_r^T v_r}
\end{aligned}$$

Its dataflow graph is in Figure 16.

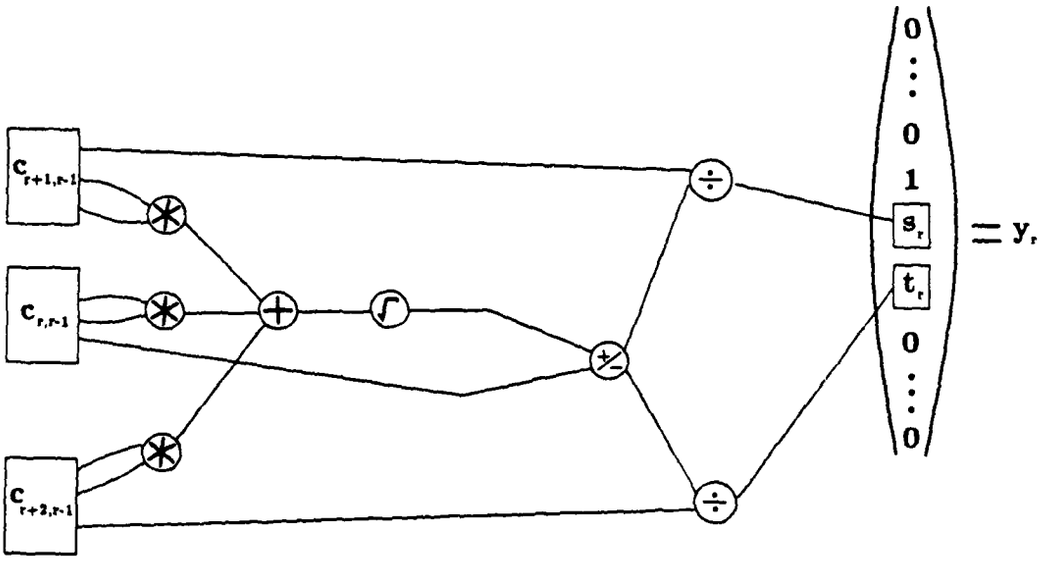


Figure 14:

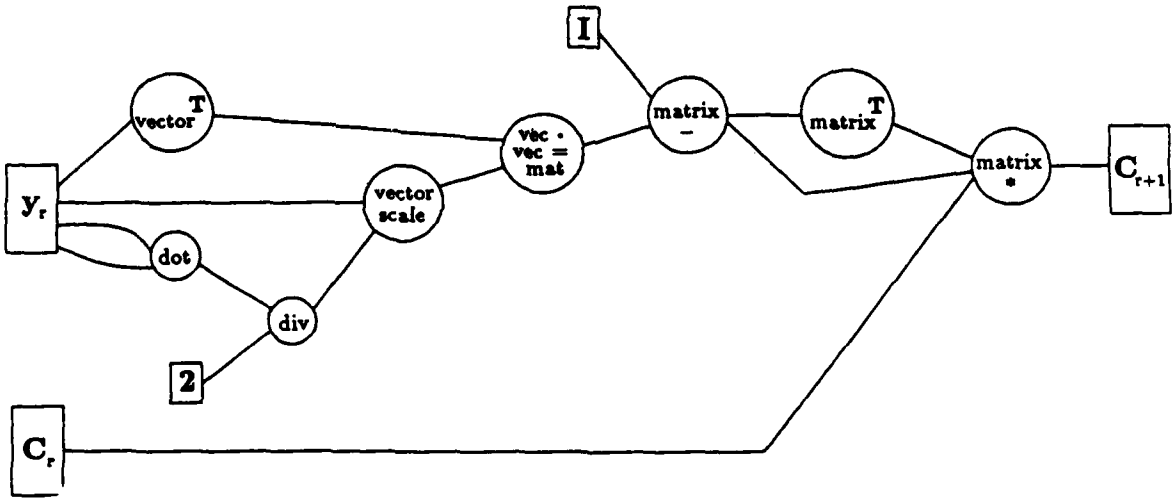

I
vector T
matrix −
matrix T
vec • vec = mat
matrix *
C_{r+1}
y_r
vector scale
dot
div
2
C_r

Figure 15:

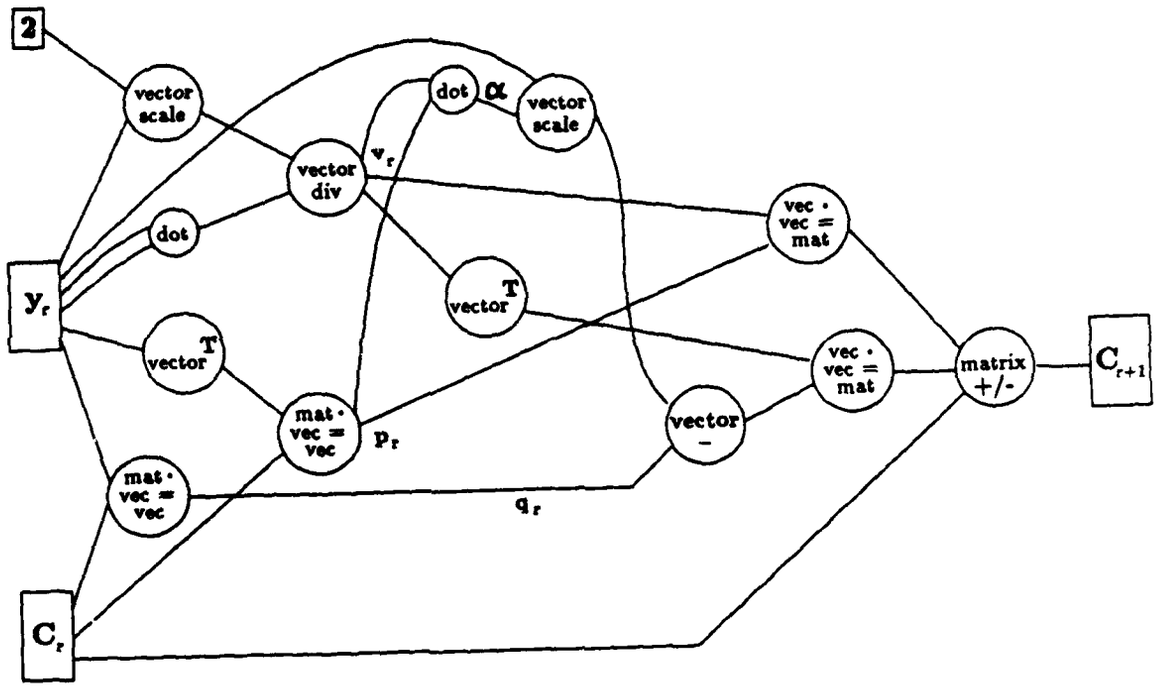

2
vector scale
dot
α
vector scale
vector div
w_r
dot
y_r
vector T
vec • vec = mat
vector T
vec • vec = mat
matrix +/-
C_{r+1}
mat • vec = vec
p_r
vector −
mat • vec = vec
q_r
C_r

Figure 16:

A.2 Householder Transforms

If a matrix representation of a system of linear equations can be transformed into an upper triangular matrix, then simple back substitution will yield the solution. Householder (HH) transforms produce upper triangular matrices ([Sch84, pages 88-90]). To convert the $N \times N$ matrix a:

For $n = 1, \ldots, N$:

$$a_{k,m}^{(n)} = a_{k,m}^{(n-1)} - \frac{1}{\sigma_n(a_{nn}^{(n-1)} + \sigma_n)} u_k^{(n)} (\sum_{i=n}^{N} u_i^{(n)} a_{im}^{(n-1)})$$

for $k = n, \ldots, N$ and $m = n, \ldots, N$ where

$$\sigma_n = sign(a_{nn}^{(n-1)}) \sqrt{\sum_{i=n}^{N} a_{in}^2}$$

and

$$\vec{u}^{(n)} = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ a_{nn}^{(n-1)} \\ \vdots \\ a_{Nn}^{(n-1)} \end{pmatrix} + \sigma_n \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

$a^{(0)}$ is the original matrix and $a^{(N)}$ is the upper triangular one.

A.3 Computing the r largest Eigenvalues

The following procedure calculates the r largest eigenvalues (EV) for a symmetric, positive definite, $n \times n$ matrix ([Sch84, pages 103-104]). Let $\mathbf{A}$ be a symmetric, positive definite matrix. Given $\mathbf{A}$ compute $\mathbf{Z_1}$ by

$$\begin{aligned} \mathbf{Z_k} &= \mathbf{A X_{k-1}} \\ \mathbf{X_k} &= \mathbf{Z_k R_k^{-1}} \end{aligned}$$

where $\mathbf{X_0}$ is an orthonormal matrix ($\mathbf{X_0^T X_0} = \mathbf{I}$) of dimension $n \times r$. Use $\mathbf{Z}$ to compute the next $\mathbf{X}$ and $\mathbf{R}$ as described below. As the number of iterations approaches infinity, the column vectors of the $\mathbf{Z}$ matrix are the eigenvectors and the diagonal elements of the $\mathbf{R}$ matrix are the corresponding eigenvalues. The eigenvalues appear in decreasing order.

$\mathbf{R_k}$ is a $r \times r$, upper triangular matrix. The termination condition requires the sum of the absolute values of the non-diagonal elements of $\mathbf{R_k}$ to be below a prescribed limit. Numerical results encourage belief in its stability. $\mathbf{X}$ and $\mathbf{R}$ are to be calculated from $\mathbf{Z}$ as follows:

Let $\mathbf{Z_k} = \mathbf{Z} = [\mathbf{z_1}, \ldots, \mathbf{z_r}]$ and $\mathbf{X_k} = \mathbf{X} = [\mathbf{x_1}, \ldots, \mathbf{x_r}]$ then

$$\begin{aligned} \mathbf{x_j} &= \frac{\mathbf{z_j} - \sum_{i=1}^{j-1} r_{ij}\mathbf{x_i}}{r_{jj}} \\ r_{ij} &= \mathbf{x_i^T z_j} \qquad for \ \ (i = 1, \ldots, j-1) \\ r_{jj} &= \sqrt{[\mathbf{z_1} - \sum_{i=1}^{j-1} r_{ij}\mathbf{x_i}]^T [\mathbf{z_j} - \sum_{i=1}^{j-1} r_{ij}\mathbf{x_i}]} \end{aligned}$$

where

$$\begin{aligned} \mathbf{x_1} &= \frac{\mathbf{z_1}}{r_{11}} \\ r_{11} &= \sqrt{\mathbf{z_1^T z_1}} \end{aligned}$$

A.4 Conjugate Gradient Method

Conjugate Gradient (CG) [PFTV87] is an indirect method to iteratively find the solution to a system of linear equations. It uses a steepest decent method where the decent directions are conjugates of each other. Let $\mathbf{Ax} = \mathbf{y}$ be a matrix representation of the equations. Given $\mathbf{A}$ and $\mathbf{y}$, guess $\mathbf{x_0}$. Then,

$$\begin{aligned} \mathbf{e_0} &= \mathbf{y} - \mathbf{Ax_0} \\ \mathbf{d_0} &= \mathbf{e_0} \end{aligned}$$

and

$$\begin{aligned} \mathbf{x_{n+1}} &= \mathbf{x_n} + \alpha_n \mathbf{d_n} \\ \alpha_n &= \frac{|\mathbf{e_n}|^2}{\mathbf{d_n^T(Ad_n)}} \\ \mathbf{e_{n+1}} &= \mathbf{e_n} - \alpha_n \mathbf{Ad_n} \\ \mathbf{d_{n+1}} &= \mathbf{e_{n+1}} + \frac{|\mathbf{e_{n+1}}|^2}{|\mathbf{e_n}|^2}\mathbf{d_n} \end{aligned}$$

The dataflow graph is shown in Figure 17.

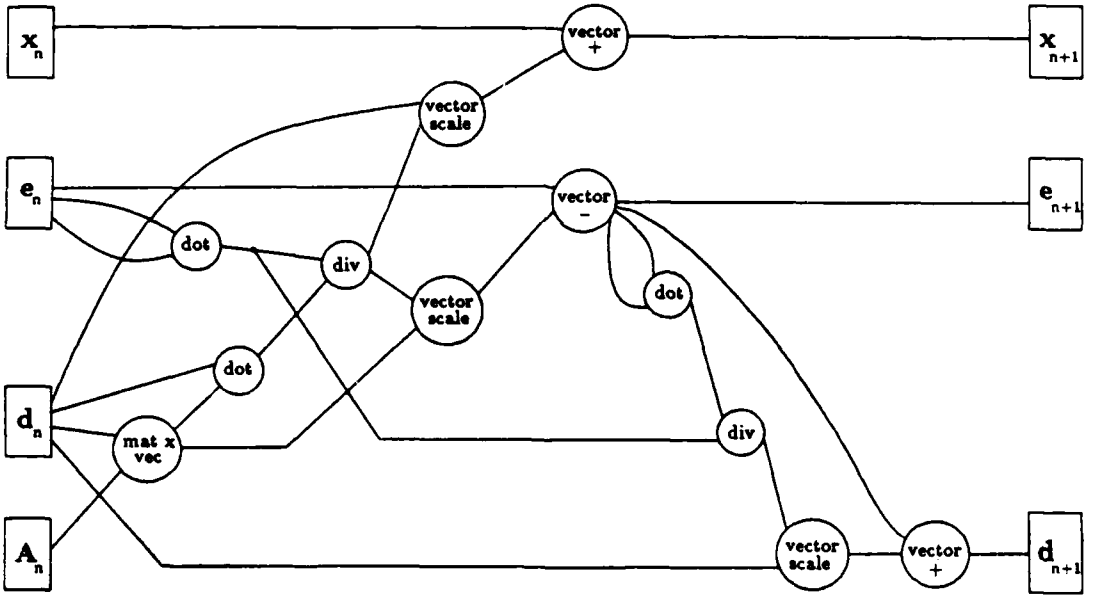

x n
vector +
x n+1
vector scale
e n
vector −
e n+1
dot
div
vector scale
dot
dot
d n
mat x vec
div
A n
vector scale
vector +
d n+1

Figure 17:

B The ABU Design System

In this appendix a proposed design system called Artificial Build Up (ABU) will be described. ABU uses the techniques from decoupled design, alteration strategies, and sample search to create a system that provides quick feedback about high quality designs. This appendix details the architectural description language, describes the program's output, and discusses user/system interaction issues.

The purpose of the implementation is to prove that the concepts can be useful in a design system; the purpose is not to build a commercially viable system that designers can actually use. If the implementation reveals limitations to expanding the system then hopefully the reasons can be elicitated and solutions suggested.

B.1 User Interaction

An unresolved issue is the extent to which the user can influence the design process. The user influences the performance characteristics of the output by adjusting his goals. But can he also influence the design process? That is, does the system allow the user language to express constraints/preferences on design steps? Various levels of influence include: the user approving the application of alteration strategies, the user recommending which alteration strategies to apply, the user approving/suggesting the disconnected design bindings, or the user expressing constraints on features of the architecture (eg: 32 bit data paths only, fast multipliers only). After the designs are created the user might want to filter the outputs. Separating the outputs by performance characteristic is straightforward, because each design is indexed by all the performance dimensions. Extracting designs that fit a structural attribute or that used a particular design process is more difficult. A query language that is specialized for this domain would have to express filters like: remove all the designs that use CCDs as operators or retrieve all the designs that use the synchronization alteration strategy. Clearly, these features would be useful, but it is not obvious whether they are necessary for establishing the utility of the design system. A decision will be made later in the implementation process.

B.2 Details of the Design System Specification

The system's input language should be expressive enough to encode key parts of heterogeneous signal processing algorithms [8]. The need for an expressive input language should be balanced by the the necessity of a reasonable implementation. This balance applies not only to the input language but for all the specifications. Here are the current specifications for the system:

[8] It does not have to encode *the entire* algorithm; for example the initialization part can be ignored.

- The user's functional input will be a restricted dataflow graph that contains only non-branching computation fragments. This is a necessary part of implementing more general computation, so it is a reasonable place to start. In later versions, adding control constructs can be done without significant interaction with the computation fragment. As the system's performance range is extended and as control constructs increase the size of the graphs, it may be necessary to map several different computation fragments into one hardware structure so computation fragments and control will interact. But, the system will be useful even at the stage where implementation of fragments and control constructs are independent.

- The input language will include operators that manipulate "high level" data types:

  - Matrix: addition, subtraction, multiplication (including matrix × vector and vector × vector), matrix scale, transpose.

  - Vector/sequence: addition, subtraction, dot product, vector scale, vector scale/accumulate, vector magnitude, magnitude squared, shift (left, right; circular, linear), summation, linear and circular convolution.

  - Scaler: addition, subtraction, multiplication, division, negation, squaring, raising to the n^{th} power, square root, maximum, minimum, absolute value, logarithm, reciprocal, increment, decrement, multiply by a power of 2, bit operations on scalers, truncate, pad with zeros.

  - Complex numbers: addition, subtraction, multiplication, division, real or imaginary part, magnitude, magnitude squared, angle, conjugate.

  - Bits: and, or, not, xor.

- Parenthesization of input expressions will be optional and the program will try different parses. The HH and the EV computations (see Appendix A) require an especially expressive language: variable length vectors and matrices, sophisticated matrix indexing, etc.

- All operators will accept only one data type: fix-width integers. Adding boolean types is straight forward and is necessary when implementing control constructs. Intelligent processing of arrays can range from nonexistent to very sophisticated. An appropriate compromise will be chosen. The ability to vary the data type along constrained dimensions (eg: both 16-bit and 20-bit integers) is straight forward and fits nicely into the flavor of the overall solution. Floating point numbers are important in signal processing. Unfortunately, floating point operators have very different internal structure from their fixed point counterparts, and the methods in this proposal rely on

knowledge about the internal structure. There is nothing about floating point operator structures that prevents them from using this method. The initial implementation will have only fixed point operators.

- Later versions of the system will include a language to describe memory access patterns so interleaved memories can be build. For the initial implementation, simple memories with fixed fetch times will be used. A single design may have several memories of different types.

- The performance specifications will consist of time (both throughput and latency), chip area, power dissipation, bit accuracy, and number of design cells types. The model for the throughput time is very simple. Each device is assigned a delay time. There is no SPICE simulation and the capacitance of the output is not considered in this model. Latency will differ from throughput because of pipelining. The chip area will be a one dimensional sum instead of a 2 dimensional description. Routing area and control circuitry will be allocated by assigning a constant fraction of the chip area to these functions. The area dimension is assumed to be the area of a single chip implementation. This may not be reasonable for larger architectures but it will give a reasonable approximation for multi-chip implementations. Power consumption is often the bottleneck in a system so it should be included. Bit accuracy is a measure of numerical accuracy of the result. No sophisticated numerical analysis will be done. Rules may be used to approximate the accuracy lost (eg: a multiplier looses top half of the result). The number of design cells gives an indication of the complexity of the design and layout. These simplifications are used by real designers ([RFS83]) and are a standard way to make the problem tractable at a high-level. If more accurate performance measurements are needed, additional detail can be added to the models. For example, circuit area could be estimated using 2 dimensional rectangles. A scheme would have to be devised to get a 2 dimensional estimate. Since the 2 dimensional method would be slower than summing one dimensional areas, a mixed strategy could use summation as a first pass and the more accurate evaluation for architectures close to the user's goal.

- The output of the program will be a set of architectures. Each architecture will implement the input's functional behavior but will differ from the others in performance. For ease of display, the program will plot all the architectures on a grid (eg: time vs area). The program is expected to automatically produce several radically different architectures for the dataflow fragment. The user should be able to select a point on the grid and see the details of the corresponding architecture. The architectural description will consist of connected functional blocks, not a layout or floorplan. It

will also include the control sequence [9] for the architecture.

- As an alternative to specifying an input expression, the user can describe his own architecture and let the program simulate it's performance. More generally the user could partially specify an architecture and let the program complete and simulate it.

These restrictions should be relaxed one by one in later versions of the system.

C A Simplification Transforms Example

The following example shows the utility of simplification transforms. The sharing alteration strategy is pictorially described in Figure 18 where resources A and B are any type of

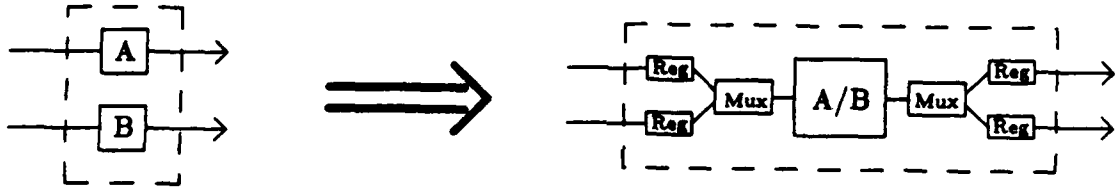
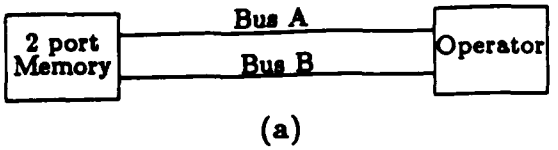


Figure 18:

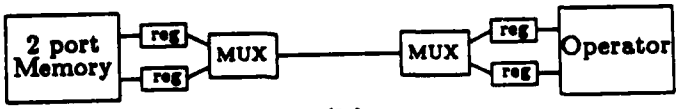
resource: operator, bus, or memory. Each resource must accept an input and produce an output as shown in the figure. In this example, A and B will be buses. Figure 19a shows a 2 port memory connected to a 2 port operator via 2 buses. If the system wanted to share the buses it would interpret them into the resources A and B of Figure 18. The knowledge base would know that buses are considered resources with 2 I/O ports. The instantiated alteration strategy would transform the unshared configuration into the shared configuration (Figure 19b).

Note that if the system implements the 2 port memory with a 1 port memory (as shown in Figure 19c), then the architecture is ripe for simplification transforms to be applied. Figure 19d shows the memory implementation spliced into the overall architecture. At this point, some set of tools needs to simplify the architecture by getting rid of redundant parts. Simplification transforms might notice that the register pairs R2/R4 and R3/R5

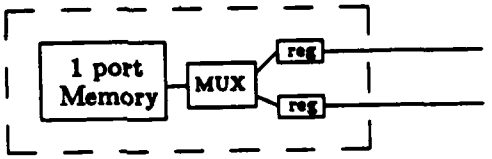
[9] The control sequence should not be confused with the control constructs in the functional specification. Control constructs refer to conditionals and loops in the input specification. The control sequence coordinates operator execution and data routing for a particular architecture.



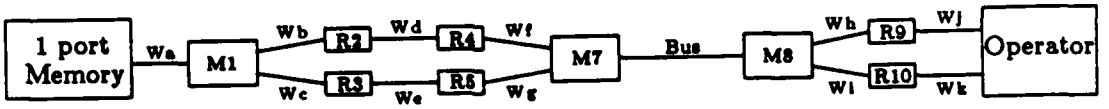
(a)



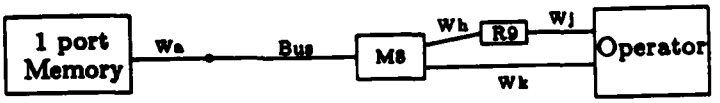
(b)



(c)



(d)



(e)

Figure 19:

are redundant since there are no operators between them. A more traditional optimizing compiler tool might look at the control flow and notice it could collapse the data transfers $Mem \rightarrow R2 \rightarrow R4 \rightarrow R9$ to $Mem \rightarrow R9$ and similarly for $Mem \rightarrow R10$. The tool would also have to modify the design to remove redundant multiplexors. (It may be necessary to use a combination of optimizing tools and simplification transforms to clean up a transformed architecture.) A final simplification rule would put the finishing touch on the architecture by removing one last register. Notice that the memory is acting as a serial source whose purpose is to simulate a parallel source (2 port memory). The serial source sequentially stores values in each register and then releases them when the last register is updated. An simplification transform can eliminate the last register by insuring that the serial source provides the last register's value. The final architecture is shown in Figure 19e.

As an aside, [Mos85] points out that a correctness-preserving transformation history is useful to verify that the design implements the specification. But, the transformation of the target design back to the source design may be more succinct. That is, 2 designs can be shown to be equal by transforming one to the other. The transformation history is one such justification, but it might be very long and/or complicated. A simpler justification might be the transform history of the reverse engineered deduction. It is just as valid as the forward history, and it maybe easier to design with. The reverse history can be used in explanation, as the foundation for building compiled empirical associations, etc. The reverse engineered justification for Figure 19e is: imagine a box around all the parts in the design except the operator. The system can interpret the box as a simulation of a 2 port memory by a one port memory. Then it is clear that Figure 19e is equivalent to Figure 19a by replacing the simulated memory with a real 2 port memory. Contrast this justification to the long winded derivation in the other direction.

Simplification transforms are similar to decoupled design's local optimizer. They both transform inefficient structures into more efficient ones. But, there are a few differences. Simplification transforms are applied to abstract designs. Since a change in an abstract design may affect many instantiated designs, all simplification transforms probably should be applied. Also, the justification of the two mechanisms is somewhat different which may lead to different reasons for apply them. Normally, simplification transforms clean up the inefficiencies left by the alteration strategies. But a simplification transform might be run in reverse with the intent of making a structure less efficient but more structured so another alteration strategy can apply. The local optimizer's transforms will not be reversed. Finally, from an implementation point of view it might be desirable to keep optimizations of abstract and concrete designs separate for efficiency.

References

[Bal81] Robert Balzer. Transformational implementation: An example. *IEEE Transactions on Software Engineering*, SE-7(1), 1981.

[BG86] Forrest D. Brewer and Daniel D. Gajski. An expert-system paradigm for design. In *Proceedings of the Design Automation Conference*, pages 62–68. Association for Computing Machinery/The Institute of Electrical and Electronics Engineers, 1986.

[CS84] Peter R. Cappello and Kenneth Steiglitz. Unifying VLSI array design with linear transformations of space-time. In *Advances in Computing Research*, volume 2, pages 23–65. JAI Press, 1984.

[Dav80] Randall Davis. Meta-rules: Reasoning about control. *Artificial Intelligence*, 15(3):179–222, 1980.

[DR85] Peter Denyer and David Renshaw. *VLSI Signal Processing: A Bit-serial Approach*. Addison-Wesley Publishing Company, 1985.

[Fog88] Dennis C. Fogg. Assisting design given performance specifications involving multiple criteria, 1988. Dissertation Proposal, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.

[Ign82] James P. Ignizio. *Linear Programming in Single- and Multiple-Objective Systems*. Prentice-Hall, Englewood Cliffs, N.J., 1982.

[Kan79] Elaine Kant. A knowledge-based approach to using efficiency estimation in program synthesis. In *Proceedings of the Sixth International Joint Conference on Artificial Intelligence*, pages 457–462, 1979.

[KB81] Elaine Kant and David R. Barstow. The refinement paradigm: The interaction of coding and efficiency knowledge in program synthesis. *IEEE Transactions on Software Engineering*, SE-7(5), 1981.

[KKP+81] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *8th Annual ACM Symposium on Principles of Programming Languages*, pages 207–218. Association for Computing Machinery, January 1981.

[Kow85] Thaddeus J. Kowalski. *An Artificial Intelligence Approach to VLSI Design*. Kluwer Academic Publishers, 1985.

[KP86] David W. Knapp and Alice C. Parker. A design utility manager: The ADAM planning engine. In *Proceedings of the Design Automation Conference*, pages 48–54. Association for Computing Machinery/The Institute of Electrical and Electronics Engineers, 1986.

[LM85] Monica S. Lam and Jack Mostow. A transformational model of VLSI systolic design. *IEEE Computer*, pages 42–52, February 1985.

[LNR87] John E. Laird, Allen Newell, and Paul S. Rosenbloom. SOAR: An architecture for general intelligence. *Artificial Intelligence*, 33:1–64, 1987.

[Mei81] J. Meier. Real time control of marco data flow architectures. In *Real-time Systems Symposium*, pages 95–98, 1981.

[Mil82] Leslie Jill Miller. A heterogeneous multiprocessor design and the distributed scheduling of its task group workload. In *9th Annual Symposium on Computer Architecture*, pages 283–290, 1982.

[Mir79] Glen S. Miranker. *The Use of Conflict in the Translation and Optimization of Hardware Description Languages.* PhD thesis, Massachussetts Institute of Technology, 1979.

[Mos85] Jack Mostow. Towards better models of the design process. *AI Magazine*, pages 44–57, Spring 1985.

[MSKC+83] Tom M. Mitchell, Louis I. Steinberg, Smadar Kedar-Cabelli, Van E. Kelly, Jeffrey Shulman, and Timothy Weinrich. An intelligent aid for circuit redesign. In *Proceedings of the National Conference on Artificial Intelligence.* American Association for Artificial Intelligence, 1983.

[MSS84] Tom M. Mitchell, Louis I. Steinberg, and Jeffrey Shulman. A knowledge-based approach to design. In *Proceedings of the IEEE Workshop on Principles of Knowledge-Based Systems*, pages 27–34, Denver, CO, December 1984.

[Nav87] Dundee J. Navinchandra. *Exploring Innovative Designs by Relaxing Criteria and Reasoning from Precedent Knowledge.* PhD thesis, Massachussetts Institute of Technology, 1987.

[New78] Allen Newell. Harpy, production systems and human cognition. Technical Report CMU-CS-78-140, Department of Computer Science, Carnegie-Mellon University, 1978.

[Pea84] Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley, 1984.

[PFTV87] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes.* Cambridge University Press, Cambridge, England, 1987.

[RFS83] G. G. Rassweiler, M. D. Fisher, and M. A. Shaver. A VHSIC digital adaptive processor using the LMS algorithm. Technical report, Harris Corporation, Government Systems Sector Operations, P.O. Box 37, Melbourne, Florida 32901, 1983.

[Roy83] Gerald Lafael Roylance. A simple model of circuit design. Technical Report AI-TR-703, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 1983.

[RSW79] C. Rich, H. E. Shrobe, and R. C. Waters. An overview of the programmer's apprentice. In *Proceedings of the Sixth International Joint Conference on Artificial Intelligence*, 1979.

[Sch84] U. Schendel. *Introduction to Numerical Methods for Parallel Computers.* Ellis Horwood Limited, 1984.

[Sou83] Jay R. Southard. MacPitts: An approach to silicon compilation. *IEEE Computer*, pages 74–82, December 1983.

[Thi79] Paul R. Thie. *An Introduction to Linear Programming and Game Theory.* John Wiley and Sons, New York, 1979.

[Ton86] Christopher Tong. Goal-directed planning of the design process. AI/VLSI Project Working Paper 34, Rutgers University, Department of Computer Science, New Brunswick, NJ 08903, September 1986. Summitted to the 3rd IEEE Conference on Artificial Intelligence Applications, Feb. 1987.

[Wel85] Michael Paul Wellman. Reasoning about preference models. TR 340, Massachussetts Institute of Technology, Laboratory for Computer Science, 545 Technology Square, Cambridge, MA, 02139, May 1985.

OFFICIAL DISTRIBUTION LIST

Director Information Processing Techniques Office Defense Advanced Research Projects Agency 1400 Wilson Boulevard Arlington, VA 22209	2 copies
Office of Naval Research 800 North Quincy Street Arlington, VA 22217 Attn: Dr. R. Grafton, Code 433	2 copies
Director, Code 2627 Naval Research Laboratory Washington, DC 20375	6 copies
Defense Technical Information Center Cameron Station Alexandria, VA 22314	12 copies
National Science Foundation Office of Computing Activities 1800 G. Street, N.W. Washington, DC 20550 Attn: Program Director	2 copies
Dr. E.B. Royce, Code 38 Head, Research Department Naval Weapons Center China Lake, CA 93555	1 copy